University of Alberta

HEURISTICS FOR INTEGER PROGRAMS

by

Shubhashis Ghosh

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-29682-0
Our file Notre référence
ISBN: 978-0-494-29682-0

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Many real world optimization problems can be formulated as mixed integer programs. In general, finding optimal, or even feasible, solutions to such problems is computationally intractable. For this reason, there is interest in developing heuristic algorithms for these problems.

In this thesis we present three new mixed integer program heuristic algorithms.

The first, PIVOT AND GOMORY CUT (PGC), is a feasibility heuristic, namely one that tries only to find a feasible solution. PGC integrates Gomory cuts into the bounded variable revised simplex pivoting framework similar to that used in the classic PIVOT AND COMPLEMENT heuristic of Balas and Martin.

The second, DISTANCE INDUCED NEIGHBOURHOOD SEARCH (DINS), is an improvement heuristic, namely one that starts with a feasible solution and tries to improve it as much as possible. DINS performs neighbourhood search at different nodes of the mixed integer program search tree where the mixed integer program search tree is the tree generated by either a branch-and-bound or a branch-and-cut solver. DINS defines the neighbourhoods by modeling a distance metric between the current mixed integer feasible solution and the relaxation solution at the node of the mixed integer program search tree.

The third, NEIGHBOURHOOD PIVOT AND GOMORY CUT (NPGC), is a 'find-and-improve' heuristic. Such a heuristic tries to find the best possible feasible solution. NPGC, an extention of PGC, uses Gomory cuts to define neighbourhoods, searches the neighbourhoods for feasible solutions, and improves any found feasible solutions by applying the LOCAL BRANCHING heuristic of Fischetti and Lodi.

We also present a new class of hard 0-1 integer programs for which instances are easy to generate pseudo-randomly. These pseudo-randomly generated instances are useful in comparing the performance of different heuristics.

Acknowledgements

Getting admission to the graduate school of University of Alberta is one of the greatest rewards and honours of my life. About five years ago I start this difficult but rewarding journey, and today I hope that I will be able to remember everyone who have helped me smoothen this difficult journey.

First and foremost, I thank my supervisor, Ryan B. Hayward, for taking me as his student, showing optimism in my research, and introducing me to some exciting problems in graph theory and combinatorial optimization. He has been an ideal supervisor in every respect, in terms of providing valuable technical inputs in my research and in terms of granting enough financial support that helped me to maintain my family. It is he who has taught me how a scientist should think of a problem and its possible solutions. His tireless editorial effort has undoubtedly improved the quality of this thesis. Working with him has become an enlightening experience and, in a nutshell, this thesis would be impossible without his help and co-operation.

I thank Professor Joe Culberson, Professor Lorna Stewart, Professor Renè Poliquin, and Professor Michael Buro for helping me to select interesting research directions to explore. I also want to thank many well known researchers who have provided useful information and advice at various stages of my research, including, but not limited to: Bill Cook, Gerard Cornuéjols, Andrea Lodi, Emilie Danna, Mikhail Nediak, Ed Rothberg, and Fadi Aloul. I also thank Neil Burch for helping me use ILOG Cplex, and all other support staff of the Computing Science Department at University of Alberta for their various contributions throughout my doctoral journey.

I like to express my endless gratitude to my parents for encouraging me over the years of my studies since childhood. I would also like to thank all of my relatives and friends who have worked as a support network over the time.

Finally, I thank my wife, Lovely, for her companionship and endless mental support toward my research, and for taking care of our daughter Sreoshi during this difficult time that I will cherish over the years.

Contents

1 Introduction 1

2 Background 4

2.1 Integer Programs 4

2.1.1 The Linear Programming Relaxation 5

2.1.2 The Simplex Method 5

2.1.3 Integer Programming Complexity 7

2.1.4 Approximation Algorithms 8

2.1.5 Exact Solvers for Integer Programs 9

2.1.6 Gomory Cuts 12

2.1.7 Pseudo-Boolean Solvers 14

2.2 Heuristics for Integer Programs 15

2.2.1 Feasibility Heuristics 15

2.2.2 Pivot and Complement and Its Successors 15

2.2.3 Octahedral Neighbourhood Enumeration 17

2.2.4 Pivot, Cut, and Dive 18

2.2.5 Convexity Cut 20

2.2.6 Vertex Cut 20

2.2.7 Tabu Search in Solving Integer Programs 20

2.2.8 Heuristics Based on Interior Path 21

2.2.9 Feasibility Pump 22

2.2.10 Improvement Heuristics 22

2.2.11 Local Branching 23

2.2.12 Relaxation Induced Neighbourhood Search 24

2.3 Benchmark Integer Program Instances 25
2.3.1 Existing Library of Integer Program Instances 25
2.3.2 Generation of Hard Integer Program Instances 26

3 Pivot and Gomory Cut **29**
3.1 Pivot and Complement 29
3.2 Pivot and Gomory Cut 34
3.3 Heuristic Performance Evaluation 45
3.4 PGC Performance Evaluation 46
3.4.1 PGC_0 versus PGC_1 versus PC 46
3.4.2 PGC_1 versus Feasibility Pump 48
3.4.3 PGC_1 versus ILOG Cplex 9.13 49
3.4.4 PGC versus a Pseudo-Boolean Solver 51
3.4.5 Performance on Randomly Generated Instances 51
3.4.6 Weakness of PGC 54
3.5 Complexity of PGC 54

4 Distance Induced Neighbourhood Search **56**
4.1 Distance Induced Neighbourhood Search 57
4.2 DINS Performance Evaluation 62
4.2.1 DINS Performance Evaluation from the Presumably Poor Solutions 63
4.2.2 DINS Performance Evaluation from the Presumably Good Solutions . . . 66
4.2.3 DINS Neighbourhoods versus RINS Neighbourhoods 73
4.2.4 Verification of Intuitions used in DINS 74
4.2.5 Performance on Randomly Generated Instances 75

5 Neighbourhood Pivot and Gomory Cut **79**
5.1 Neighbourhood Pivot and Gomory Cut 80
5.2 NPGC Performance Evaluation 82

6 Generating Hard Integer Program Instances **88**
6.1 Cornuéjols-Dawande Feasibility-Hard Instances 88
6.2 Constrained Williams's Market-Sharing Problems 91
6.2.1 The Expected Number of Solutions 93
6.2.2 Probability of Generating Infeasible Instances 95

6.3 Solver Performance on Constrained Market-Sharing Instances 97

7 Conclusions **101**

Bibliography **105**

Index **109**

A Appendix: Pseudo-Code of PGC **110**

B Appendix: Experimental Results **118**

B.1 Benchmark Instances 118

B.2 PGC Experimental Results 123

B.3 DINS Experimental Results 140

B.4 NPGC Experimental Results 152

List of Tables

3.1 PC versus PGC_0 on benchmark instances 47

3.2 PC versus PGC_1 on benchmark instances 47

3.3 PGC_0 versus PGC_1 on benchmark instances 48

3.4 FP versus PGC_1 on benchmark instances 49

3.5 Cplex-D versus PGC_1 on benchmark instances 50

3.6 Cplex-F versus PGC_1 on benchmark instances 50

3.7 PBS4 versus PGC_1 on benchmark instances 51

3.8 Probability measures for the Cornuéjols-Dawande feasibility-hard instances 52

4.1 Cplex-D versus DINS on benchmark instances from poor solutions 64

4.2 LB versus DINS on benchmark instances from poor solutions 65

4.3 RINS versus DINS on benchmark instances from poor solutions 65

4.4 The average and the standard deviation of percentage of gaps obtained by Cplex-D, LB, RINS, and DINS on benchmark instances from poor solutions 65

4.5 The average and the standard deviation of percentage of improvements obtained by Cplex-D, LB, RINS, and DINS on benchmark instances from poor solutions . . . 66

4.6 Cplex-D versus DINS on benchmark instances from good solutions 69

4.7 LB versus DINS on benchmark instances from good solutions 69

4.8 RINS versus DINS on benchmark instances from good solutions 69

4.9 The average and the standard deviation of percentage of gaps obtained by Cplex-D, LB, RINS, and DINS on benchmark instances from good solutions 70

4.10 The average and the standard deviation of percentage of improvements obtained by Cplex-D, LB, RINS, and DINS on benchmark instances from good solutions . . . 70

4.11 Cplex-D versus DINS on Cornuéjols-Dawande optimality-hard instances 75

4.12 LB versus DINS on Cornuéjols-Dawande optimality-hard instances 76

4.13 RINS versus DINS on Cornuéjols-Dawande optimality-hard instances 76
4.14 The average and the standard deviation of percentage of improvements obtained by Cplex-D, LB, RINS, and DINS on Cornuéjols-Dawande optimality-hard instances 76
4.15 Cplex-D versus DINS on constrained market-sharing instances 77
4.16 LB versus DINS on constrained market-sharing instances 78
4.17 RINS versus DINS on constrained market-sharing instances 78
4.18 The average and the standard deviation of percentage of improvements obtained by Cplex-D, LB, RINS, and DINS on constrained market-sharing instances 78

5.1 Cplex-D versus NPGC on benchmark instances 85
5.2 LB versus NPGC on benchmark instances 86
5.3 RINS versus NPGC on benchmark instances 86
5.4 The average and the standard deviation of percentage of gaps obtained by Cplex-D, LB, RINS, and NPGC on benchmark instances 86

6.1 Probability measures for the Cornuéjols-Dawande feasibility-hard instances 89
6.2 Probability measures for the constrained market-sharing instances 95
6.3 Different solvers on constrained market-sharing instances in finding a feasible solution 98

B.1 All mixed integer program instances from MIPLIB 2003 118
B.2 All mixed integer program instances from DEIS operations research library 121
B.3 15 new 0-1 mixed integer program instances used in [29] 122
B.4 PC, PGC_0, and PGC_1 on benchmark instances 123
B.5 FP and PGC_1 on benchmark instances 126
B.6 GLPK 4.0 versus Cplex 9.13 linear programming solvers 129
B.7 Cplex-D, Cplex-F, and PGC_1 on benchmark instances 131
B.8 PBS4 and PGC_1 on benchmark instances 134
B.9 Different solvers on Cornuéjols-Dawande feasibility model instances 134
B.10 Different solvers on constrained market-sharing instances generated with $k = 2.0$. 135
B.11 Different solvers on constrained market-sharing instances generated with $k = 1.5$. 137
B.12 Different solvers on constrained market-sharing instances generated with $k = 1.3$. 139
B.13 Percentage of gaps obtained by Cplex-D, LB, RINS and DINS in one CPU-hour on benchmark instances from poor solutions 140
B.14 Cplex-D, LB, RINS, and DINS on Cornuéjols-Dawande optimality-hard instances 143

B.15 Cplex-D, LB, RINS, and DINS on pseudo-randomly generated constrained market-sharing instances with $k = 2.0$ 146
B.16 Cplex-D, LB, RINS, and DINS on pseudo-randomly generated constrained market-sharing instances with $k = 1.5$ 147
B.17 Cplex-D, LB, RINS, and DINS on pseudo-randomly generated constrained market-sharing instances with $k = 1.3$ 149
B.18 RINS neighbourhoods versus DINS neighbourhoods 150
B.19 Percentage of gaps obtained by Cplex-D, LB, RINS, and NPGC in one CPU-hour on benchmark instances 152

List of Figures

2.1 A linear program instance 6

2.2 A simplex tableau 6

2.3 A dictionary representation 7

2.4 Algorithm branch-and-bound 10

2.5 Algorithm branch-and-cut 11

2.6 Illustration of K and K^* in two-dimension 17

2.7 Algortihm LB 24

2.8 Algorithm RINS 25

3.1 Algorithms PGC_0 and PGC_1 37

3.2 Illustration of PGC_1 on a small example 43

4.1 Algorithm DINS 61

4.2 Average percentage of gap on the small spread instances from poor solutions 67

4.3 Average percentage of gap on the medium spread instances from poor solutions . . 67

4.4 Average percentage of gap on the large spread instances from poor solutions 68

4.5 Average percentage of gap on the small spread instances from good solutions . . . 71

4.6 Average percentage of gap on the medium spread instances from good solutions . . 72

4.7 Average percentage of gap on the large spread instances from good solutions 72

5.1 Algorithm NPGC 83

Chapter 1

Introduction

Many real world optimization problems can be formulated as integer programs. Some such problems include airline crew scheduling, vehicle routing, and production planning. Consider for example the airline crew scheduling problem described by Hoffman and Padberg [44].

In airline crew scheduling, the problem is to determine the schedules for the crews from the given schedules of flights. In finding a feasible schedule, one has to confirm all the regulations and requirements set by the aviation administration, the union, and the company. Satisfying these requirements a feasible rotation for a flight is identified as the sequence of flight segments starting and stopping at particular base locations. For each of the feasible rotations, there is an associated cost. Hoffman and Padberg [44] showed that given a set of feasible rotations, the problem of minimizing the cost while finding a collection of rotations that cover each flight segment by exactly one rotation can be formulated as the following integer programming problem:

$$\begin{aligned} \min \quad & \textstyle\sum_{j=1}^{n} c_j x_j \\ \text{s.t.} \quad & \textstyle\sum_{j=1}^{n} a_{ij} x_j = 1 \quad i = 1, \ldots, m, \\ & x_j \in \{0,1\} \qquad\quad j = 1, \ldots, n, \end{aligned}$$

where n is the number of considered feasible rotations, m is the number of flight segments, a_{ij} is one if flight segment i is covered by rotation j and zero otherwise, c_j is the cost associated with rotation j, and x_j is a binary variable which is one if rotation j is selected and zero otherwise.

Observing the practical importance of integer programs, scientists have developed many solution methods for these problems. Exact methods are devised to find an optimal solution, possibly with a certificate of optimality. Heuristic methods are devised to find the best possible solution that can

be found with some reasonable amount of time. Heuristic methods give no guarantee of finding a feasible solution, and there is no guarantee that any found feasible solution will be optimal.

Practitioners may not be interested in exact methods mainly for two reasons. First, the known exact methods may be computationally inadequate within the available computation time for the given size of problems. Second, finding an optimal solution may not deserve the effort required to find it. In such cases, practitioners look for heuristic methods.

Roughly the existing heuristics for integer programs can be categorized into two classes. Feasibility heuristics try only to find a feasible solution. Improvement heuristics try to find improved feasible solutions from a known feasible solution.

This thesis focuses on designing heuristic methods for integer programs and on generating hard integer program instances that can be used as benchmarks for evaluating the performance of integer program heuristics. This thesis has four major contributions.

The first contribution is a new feasibility heuristic PIVOT AND GOMORY CUT (PGC). This heuristic integrates Gomory cuts [41] into a bounded variable revised simplex pivoting framework similar to that used in PIVOT AND COMPLEMENT (PC) heuristic of Balas and Martin [13].

The second contribution is a new improvement heuristic DISTANCE INDUCED NEIGHBOURHOOD SEARCH (DINS). This heuristic performs neighbourhood search at different nodes of the mixed integer program search tree where the mixed integer program search tree is the tree generated by either a branch-and-bound or a branch-and-cut solver. DINS defines neighbourhoods by modeling a distance metric between the current mixed integer feasible solution and the relaxation solution at the node of the mixed integer program search tree.

The third contribution is a new 'find-and-improve' type heuristic NEIGHBOURHOOD PIVOT AND GOMORY CUT (NPGC). This heuristic, which is an extension of PGC, repeatedly seeks a feasible solution and tries to improve it by incorporating neighbourhood search. NPGC uses Gomory cuts to define the neighbourhoods, searches the neighbourhoods for feasible solutions employing an exact solver, and improves any found feasible solution by applying the LOCAL BRANCHING (LB) heuristic of Fischetti and Lodi [35].

The fourth contribution is a new class of 0-1 integer program instances for which finding even a feasible solution is hard. This class of instances originates from a modified form of Williams's [73] market-sharing problem.

As a final remark, Gomory cuts, although regarded since their introduction as theoretically sig-

nificant, were not recognized for their practical importance until the 1990s, when Balas, Ceria, Cornuéjols and Natraj used Gomory cuts to strengthen branch-and-bound solvers [11, 26]. By introducing new successful mixed integer programming heuristics PGC and NPGC, this thesis show that Gomory cuts can also be used to design effective integer programming heuristics.

Chapter 2

Background

2.1 Integer Programs

An *integer program* is a problem of the form

$$\min\{cx \mid x \in X\} \text{ such that } X = \{x \mid Ax \geq b, x \in \mathbb{Z}^n_+\},$$

where $\mathbb{Z}^n_+$ is the set of n-dimensional vectors of nonnegative integers. In this definition, c is a row vector of dimension n, x is a column vector of n variables, A is an $m \times n$ matrix, and b is a column vector of dimension m. Usually cx is called the *objective function* or *cost function* and $Ax \geq b$ is called the *set of constraints*.

A *0-1 integer program* is an integer program where all the variables are constrained to be either zero or one. A *mixed integer program* is an integer program where for some variables, the constraint that the variable is integer is relaxed to allow the variable to be real. A *0-1 mixed integer program* is a mixed integer program where all the integer variables are constrained to be either zero or one.

In this chapter, we review the relevant background material on integer programming. For more on integer programming, see the book 'Integer and Combinatorial Optimization' by Nemhauser and Wolsey [62], 'Theory of Linear and Integer Programming' by Schrijver [67], 'Combinatorial Optimization' by Cook, Cunningham, Pulleyblank, and Schrijver [25], 'Integer Programming' by Wolsey [74].

2.1.1 The Linear Programming Relaxation

The *linear programming relaxation* of a given integer program is a program where for the variables, the constraint that the variable can only take integer values in a given interval is relaxed to allow the variable to take real values in that interval.

The linear programming relaxation of the integer program

$$\min\{cx \mid x \in X\} \text{ such that } X = \{x \mid Ax \geq b, x \in \mathbb{Z}_+^n\}$$

is the problem

$$\min\{cx \mid x \in X\} \text{ such that } X = \{x \mid Ax \geq b, x \in \mathbb{R}_+^n\},$$

where $\mathbb{Z}_+^n$ and $\mathbb{R}_+^n$ are the sets of n-dimensional vectors of nonnegative integers and nonnegative real numbers respectively.

Similarly, the linear programming relaxation of the 0-1 integer program

$$\min\{cx \mid x \in X\} \text{ such that } X = \{x \mid Ax \geq b, x \in \{0,1\}^n\}$$

is the problem

$$\min\{cx \mid x \in X\} \text{ such that } X = \{x \mid Ax \geq b, 0 \leq x \leq 1\}.$$

Mixed integer programs also have corresponding linear programming relaxations.

The solution space of an integer program is a subset of the solution space of its linear programming relaxation. Therefore, the usual starting point to solve an integer program is to solve its linear programming relaxation. In the best case, the optimal solution of the linear programming relaxation is integer and so an optimal solution to the integer program.

2.1.2 The Simplex Method

The simplex method, due to Dantzig [30], is a popular method for solving linear programming problems.

In its first step, the simplex method introduces the so-called surplus variables to convert the given constraints in the inequality form to the form of equations. For example, the simplex method introduces a surplus variable y_i for each constraint $A_i x \geq b_i$ in the problem

$$\min\{cx \mid x \in X\} \text{ such that } X = \{x \mid A_i x \geq b_i \quad \forall i \in \{1 \cdots m\},\ x \in \mathbb{R}_+^n\}.$$

The modified problem has the form

$$\min\{cx \mid x \in X\} \text{ such that } X = \{x \mid A_i x - y_i = b_i \quad \forall i \in \{1 \cdots m\},\ x \in \mathbb{R}_+^n,\ y_i \geq 0\}.$$

If the given problem in inequality form has n variables and m constraints, the modified problem has $n + m$ variables. The *basic variables* are the m variables corresponding to any chosen m linearly independent columns. The *nonbasic* variables are the remaining n variables. A *basis* is

the submatrix in the constraint matrix of the modified problem considering the coefficient of basic variables. The solution comprising the value of basic variables, evaluated by setting the nonbasic variables to zero[1], is a *basic solution*. If a basic solution is feasible for the given linear program, it is called a *basic feasible solution*.

After introducing the surplus variables, the simplex method finds a basic feasible solution to start with. It then generates successive basic feasible solutions by the so-called pivoting operation until it finds a basic feasible solution with the optimal objective value. *Pivoting* is an operation where a nonbasic variable becomes a basic variable and vice versa. At each step, the simplex method chooses the pivoting nonbasic and basic variables with the aim of improving the objective value. For each basic feasible solution, a *simplex tableau* is used to represent the information necessary to carry out the next pivoting operation. For example, for the linear program instance shown in Figure 2.1,

$$
\begin{array}{ll}
\text{min} & x_1 + x_2 \\
\text{such that} & \\
& x_1 + 2x_2 + x_3 = 4 \\
& 3x_1 - 2x_2 - x_4 = 9 \\
& x_1, x_2, x_3, x_4 \geq 0
\end{array}
$$

Figure 2.1: A linear program instance

a corresponding simplex tableau is shown in Figure 2.2.

x_1	x_2	x_3	x_4	
1	1	0	0	0
1	2	1	0	4
-3	2	0	1	−9

Figure 2.2: A simplex tableau considering x_3 and x_4 as the basic variables. The variables x_1, x_2, x_3, x_4 index the columns. The top row corresponds to the objective function; the remaining rows show the coefficients of the variables in the constraint matrix where the coefficients of basic variables form an identity matrix. The rightmost column represents the value of the basic variables and the corresponding objective value.

People also use the so-called *dictionary* to represent the information of a simplex tableau. Figure 2.3 shows a dictionary representation of the simplex tableau shown in Figure 2.2.

The simplex method, when implemented so that the entire tableau is updated after each pivoting operation, is known as the *standard simplex method*. Generating successive basic feasible solutions

[1]Nonbasic variables are set to either at its lower bound or at its upper bound. In both cases we can substitute the variables with new variables in such way that the new variables have value zero. If a nonbasic variable x_i is set to its nonzero lower bound l_i, then we can replace x_i by $l_i + x'_i$ and thus x'_i is set to zero. Similarly, if a nonbasic variable x_i is set to its finite upper bound u_i, then we can replace x_i by $u_i - x'_i$ and thus x'_i is set to zero.

actually requires only a small part of the simplex tableau. The simplex method, when implemented so that this small part of the tableau required for pivoting is generated from the original data, is known as the *revised simplex method*. The revised simplex method, when implemented to handle variables with explicit bounds instead of only nonnegative variables, is known as the *bounded variable revised simplex method*. For a detailed description of these three methods, see the book 'Linear Programming' by Chvátal [23].

$$\begin{aligned} x_1 + x_2 &= C \\ 4 - x_1 - 2x_2 &= x_3 \\ -9 + 3x_1 - 2x_2 &= x_4 \end{aligned}$$

Figure 2.3: A dictionary considering x_3 and x_4 as the basic variables. C denotes the objective function.

The simplex method, for the existing pivot rules, takes an exponential number of pivot steps, in terms of number of variables, in some instances. Klee and Minty [51] showed this for Dantzig's pivot rule; Avis and Chvátal [6] showed for Bland's pivot rule; Jeroslow [46] showed for the best improvement pivot rule; Goldfarb and Sit [39] showed for the steepest edge rule. In spite of this negative behaviour, it is the simplicity and practical success of the simplex method that have established it as a popular method for linear programming. Khachiyan's ellipsoid method [49, 50] is the first method introduced to solve linear programs in polynomial time. But it is Karmarkar's interior point method [47] that provides the practical success with the polynomial complexity. This practical success has come far later from the time it was introduced by Karmarkar and mostly due to Mehrotra [59]. At present commercial optimization software such as Cplex and Xpress come with both the simplex method and the interior point method for solving linear programs.

2.1.3 Integer Programming Complexity

Following Karp's [48] reduction from a satisfiability problem to an integer program, in this section, we see that integer programs are NP-hard in general. It is immediate if we see that the decision problem corresponding to an integer program lies in the class of NP-complete problems.

The decision problem corresponding to an integer program is as follows:

Instance : A set of n integer variables $x = \{x_1, x_2, \cdots, x_n\}$, a set of m linear inequalities $Ax \geq b$, an objective function cx and an integer v.

Query : Is there a feasible solution for which the objective value is at most v?

The satisfiability problem is NP-complete [36], and it has the following form.

Instance : A set of n boolean variables $\{x_1, x_2, \cdots, x_n\}$ and a set of m clauses $\{C_1, C_2, \ldots, C_m\}$ composed of boolean variables.

Query : Is there a truth assignment, a set of values for the variables, that satisfies all the clauses?

The decision version of integer program is in the class of NP. Karp [48], in 1972, shows that there is a polynomial time reduction from any satisfiability instance to a decision version of integer program instance. The reduction goes as follows. For each variable x_i in the satisfiability instance, there is a corresponding variable x_i in the decision version of integer program instance. The decision version of integer program instance has the following set of linear constraints.

(i) $x_i \in \{0, 1\}$ for $i = 1 \ldots n$.

(ii) For each clause C_j of the satisfiability instance, suppose O_j and X_j are the set of original and complemented variables respectively. Then the corresponding constraint in the decision version of integer program is

$$\sum_{x_i \in O_j} x_i + \sum_{\overline{x_i} \in X_j} (1 - x_i) \geq 1.$$

The objective function can be an arbitrary function of the variables, for example, $\sum_{i=1}^{n} x_i$. And v has to be set considering the objective function so that any truth assignment satisfying the satisfiability problem provides a corresponding solution for the reduced decision version of integer program and vice versa. For example, v can be set to n for the objective function $\sum_{i=1}^{n} x_i$.

Therefore, following the existence in the class of NP and the reduction shown by Karp, the decision problem corresponding to an integer program is NP-complete.

The integer program instances obtained from satisfiability instances are 0-1 integer programs, which constitute a small subset of all possible integer programming instances. But, as we have seen that some of the integer programs are NP-complete, we can say that integer programs are NP-hard to solve in general.

2.1.4 Approximation Algorithms

An approximation algorithm finds a near-optimal solution of an optimization problem within a guaranteed factor of optimal solution in polynomial time where the polynomial is bounded by the size of input instance and a fixed error.

Whenever an optimization problem occurs for which finding an optimal solution is NP-hard, one may consider looking for an approximation algorithm. However, not every NP-hard problem

has an approximation algorithm. Some of these problems such as the knapsack problem, the vertex cover problem have approximation algorithms, while some such as the general traveling salesman problem, not restricted to maintain triangular inequality property, cannot have an approximation algorithm [68]. Interestingly, we can model problems of both categories as integer programs.

Therefore, in general, integer programs cannot have an approximation algorithm unless P=NP since otherwise the general traveling salesman problem would have an approximation algorithm. And if the general traveling salesman problem has an approximation algorithm, then there will be a polynomial time algorithm for finding a Hamiltonian cycle of a graph [68], which is known to be NP-complete [36].

2.1.5 Exact Solvers for Integer Programs

A complete enumeration of the solution space is a straightforward way to find the optimal solution of an integer program, but it is computationally impractical even for a moderate size problem. For example, an integer program with 100 variables, where each variable has two possible integer values, has an enumeration space of size 2^{100}. For such problems some bound information on the objective function can be useful in the process of enumeration. For an integer program where an objective function has to be minimized, a feasible solution provides an upper bound for the objective function, and the optimal solution of a relaxation of the integer program provides a lower bound for the objective function. Branch-and-bound algorithms use this bound information to prune some branches of the enumeration tree without exploration. In 1960 Land and Doig [52] presented a branch-and-bound algorithm for integer programming. Figure 2.4 gives a description of a standard branch-and-bound algorithm based on the linear programming relaxation of integer program. This pseudo-code replicates the flowchart given by Wolsey [74] describing a branch-and-bound algorithm.

In the pseudo-code, $\mathcal{S}^i$ denotes the i-th node of the branch-and-bound tree, and $\mathcal{P}^i$ denotes the formulation of the integer program at $\mathcal{S}^i$. Therefore, $\mathcal{S}^0$ denotes the root node of the branch-and-bound tree, and $\mathcal{P}^0$ denotes the initial formulation of the given integer program P. The only difference between $\mathcal{P}^0$ and $\mathcal{P}^i$ is that the lower and upper bounds for some of the variables in $\mathcal{P}^i$ are different from that in $\mathcal{P}^0$. A node list $\mathcal{L}$ stores the nodes $\mathcal{S}$. Z_u and Z_l denote the current upper bound and current lower bound of the objective function respectively. The variable x_{ip} represents the best feasible solution obtained so far.

Preprocessing the given problem, applying effective heuristics at different nodes of the tree,

choosing different nodes to explore at Step 3, and choosing different variables to branch at Step 12 affect the efficiency of this standard branch-and-bound algorithm.

Algorithm branch-and-bound
INPUT: an integer program P with formulation $\mathcal{P}^0$
OUTPUT: an optimal solution x_{ip} of P if exists, else null.

1. create node $\mathcal{S}^0$ with formulation $\mathcal{P}^0$ and put it in $\mathcal{L}$; $Z_u \leftarrow \infty$; $x_{ip} \leftarrow$ null.
2. while ($\mathcal{L} \neq \phi$)
3. choose a node $\mathcal{S}^i$ from the list $\mathcal{L}$ and delete the node from $\mathcal{L}$.
4. formulate the linear programming relaxation $\mathcal{L}(\mathcal{P}^i)$ of $\mathcal{P}^i$.
5. solve $\mathcal{L}(\mathcal{P}^i)$.
6. if ($\mathcal{L}(\mathcal{P}^i)$ has a solution)
7. $x^* \leftarrow$ the solution of $\mathcal{L}(\mathcal{P}^i)$; $Z_l \leftarrow$ objective value corresponding to x^*.
8. if ($Z_u > Z_l$)
9. if (x^* is a feasible solution for P)
10. $Z_u \leftarrow Z_l$; $x_{ip} \leftarrow x^*$.
11. else
12. choose a variable i which has fractional value in x^*.
13. create two new nodes by formulating two subproblems corresponding to $x_i \leq \lfloor x_i^* \rfloor$ and $x_i \geq \lceil x_i^* \rceil$.
14. add the two new nodes to the list $\mathcal{L}$.
15. return x_{ip}.

Figure 2.4: Algorithm branch-and-bound, a pseudo-code version of the algorithm presented by Wolsey [74].

Branch-and-bound works better if the description of the underlying feasible region corresponding to the linear programming relaxation of integer program is made tighter. The concept of generating cutting planes comes from this viewpoint. A *cutting plane* is an inequality that is satisfied by all the feasible solutions of the integer program but cuts off some region of the underlying feasible region corresponding to the linear programming relaxation of integer program. If a branch-and-bound algorithm generates cutting planes at the nodes of a branch-and-bound tree, then it is known as a branch-and-cut algorithm. In 1983 Crowder, Johnson, and Padberg [28] presented a branch-and-cut algorithm for solving 0-1 integer programs.

Adding a different number of cuts at a node and selecting different nodes to generate cuts give different implementations of a branch-and-cut algorithm. Using the notations used in the branch-and-bound pseudo-code, Figure 2.5 shows a possible way of implementing the branch-and-cut algorithm. This pseudo-code replicates the flowchart given by Wolsey [74] describing a branch-and-cut algorithm.

There are many ways to generate cutting planes for integer programs. In this thesis, we use the mixed integer cuts of Gomory [41] since we can generate these cuts easily from the simplex tableau.

We see a brief description of Gomory mixed integer cuts in the next section. There are many other cuts that have been studied in the context of solving integer programs, such as the fractional cuts of Gomory [41], the Chvátal-Gomory cuts of Chvátal [21], the convexity cuts of Balas [7], the disjunctive cuts of Balas [8], and the lift-and-project cuts of Balas, Ceria, and Cornuéjols [9, 10].

Algorithm branch-and-cut
INPUT: an integer program P with formulation $\mathcal{P}^0$
OUTPUT: an optimal solution x_{ip} of P if exists, else null.

```
1.  create node S^0 with formulation P^0 and put it in L;  Z_u ← ∞;  x_ip ← null.
2.  while (L ≠ φ)
3.      choose a node S^i from the list L and delete the node from L.
4.      formulate the linear programming relaxation L(P^i) of P^i.
5.      isFeasibleLP ← true;  generateCut ← true.
6.      repeat
7.          solve L(P^i).
8.          if (L(P^i) has a feasible solution)
9.              x* ← the solution of L(P^i).
10.             try to generate a cut that cut off x*.
11.             if(a cut is found)
12.                 add this cut to L(P^i) and consider the modified L(P^i) as L(P^i).
13.             else generateCut ← false.
14.         else isFeasibleLP ← false.
15.     until (not generateCut or not isFeasibleLP)
16.     if (isFeasibleLP)
17.         Z_l ← the objective value corresponding to x*.
18.         if (Z_u > Z_l)
19.             if (x* is a feasible solution for P)
20.                 Z_u ← Z_l;  x_ip ← x*.
21.             else
22.                 choose a variable i which has fractional value in x*.
23.                 create two new nodes by formulating two subproblems
                        corresponding to x_i ≤ ⌊x_i*⌋ and x_i ≥ ⌈x_i*⌉.
24.                 add the two new nodes to the list L.
25. return x_ip.
```

Figure 2.5: Algorithm branch-and-cut, a pseudo-code version of the algorithm presented by Wolsey [74]

Other than branch-and-bound and branch-and-cut algorithms, there are other common techniques such as Lagrangian relaxation, column generation [74] that make solving integer programs with some particular structure easier. Generalized basis reduction, proposed by Lovász and Scarf [53], is another method for solving integer programs. Cook, Rutherford, Scarf, and Shallcross [24], Wang [70], and Aardal, Bixby, Hurkens, Lenstra, and Smeltink [1] have successfully applied this method on the problems with less than 100 variables. The integral basis method is an exact algorithm introduced by Haus, Köppe, and Weismantel [42] that requires an integer program feasible solution

to start with. It has not received much attention in practice because of its limited experimental analysis.

2.1.6 Gomory Cuts

In 1960 Gomory introduced mixed integer cuts [41], now commonly referred to as Gomory cuts.

To write the formula for generating Gomory cuts, we define some notation. Let x^* be a basic feasible solution of the linear programming relaxation $\mathcal{L}(P)$ of a given mixed integer program P. Also assume that x^* is not a feasible solution for P. Let B and NB be the respective indices of basic and non-basic variables of x^* and let j be the index of a basic integer-constrained variable which has a non-integer value in x^*. Let the row corresponding to x_j in the simplex tableau have the form $x_j^* = x_j + \sum_{k \in NB} a_{jk} x_k$. Let $f_k = a_{jk} - \lfloor a_{jk} \rfloor$, $f_j = x_j^* - \lfloor x_j^* \rfloor$, and let N_I, N_C be the indices of the respective integer-constrained non-basic and continuous non-basic variables of x^*. Then the Gomory cut for x_j with respect to P and the given tableau is the inequality

$$\sum_{\substack{k \in N_I \wedge \\ f_k \leq f_j}} f_k x_k + \sum_{\substack{k \in N_I \wedge \\ f_k > f_j}} \frac{f_j(1-f_k)}{1-f_j} x_k + \sum_{\substack{k \in N_C \wedge \\ a_{jk} < 0}} \frac{-f_j a_{jk}}{1-f_j} x_k + \sum_{\substack{k \in N_C \wedge \\ a_{jk} \geq 0}} a_{jk} x_k \quad \geq \quad f_j.$$

We now give Gomory's proof that this cutting plane is a valid inequality for P and cuts off the basic feasible solution x^*.

The selected row $x_j^* = x_j + \sum_{k \in NB} a_{jk} x_k$ corresponding to the integer-constrained variable x_j is equivalent to $x_j = x_j^* + \sum_{k \in NB} a_{jk}(-x_k)$. Since x_j is integer-constrained,

$$0 \equiv x_j^* + \textstyle\sum_{k \in NB} a_{jk}(-x_k) \pmod 1.$$

Since x_j^* is not an integer, we can reduce it to its smallest possible positive fractional value f_j by adding the congruence relation $0 \equiv -\lfloor x_j^* \rfloor \pmod 1$ to the previous congruence relation. Thus,

$$0 \equiv f_j + \textstyle\sum_{k \in NB} a_{jk}(-x_k) \pmod 1.$$

Alternatively,

$$\textstyle\sum_{k \in NB} a_{jk} x_k \equiv f_j \pmod 1.$$

The rest of the proof refers to this congruence relation. If the left-hand side of the congruence relation is positive then the left-hand side differs from f_j by an integer amount and so is equal to $f_j + t$ for some nonnegative integer t. Thus,

$$\textstyle\sum_{k \in NB} a_{jk} x_k \geq f_j.$$

Since all the variables x_k are constrained to be nonnegative,

$$\sum_{k \in NB^+} a_{jk} x_k \geq f_j,$$

where $NB^+ = \{k | a_{jk} \geq 0, \forall k\}$. On the other hand, if the left-hand side of the congruence relation is negative, then the left-hand side is equal to $-t + f_j$ for some positive integer t. Thus,

$$\sum_{k \in NB} a_{jk} x_k \leq -1 + f_j.$$

Since all the variables x_k are constrained to be nonnegative,

$$\sum_{k \in NB^-} a_{jk} x_k \leq -1 + f_j,$$

where $NB^- = \{k | a_{jk} < 0, \forall k\}$. Multiplying the above inequality by the negative number $f_j/(-1 + f_j)$ we get

$$\sum_{k \in NB^-} (\frac{f_j}{-1+f_j}) a_{jk} x_k \geq f_j.$$

Since $f_j \neq 0$, the left-hand side of the congruence relation is either positive or negative. Therefore, between the two inequalities obtained by considering the left-hand side of the congruence relation as positive and negative, one has to be valid. Since, in any feasible solution the left-hand side of both the inequalities is nonnegative, we have

$$\sum_{k \in NB^+} a_{jk} x_k + \sum_{k \in NB^-} (\frac{f_j}{-1+f_j}) a_{jk} x_k \geq f_j.$$

Thus this inequality is satisfied by every P-feasible solution. And at the same time, it cuts off the current basic feasible solution x^* of $\mathcal{L}(P)$, since, at x^*, the left-hand side evaluates to zero which is smaller than f_j.

Though this is a valid cut, we can strengthen it by reducing the coefficients of variables in the inequality. Gomory does this for the coefficients of integer-constrained variables. Adding or subtracting an integer multiple of an integer-constrained variable to the left-hand side of the congruence relation yields another valid relation. Therefore, there is the option of choosing which integer-constrained variable indices should be put in the set NB^+ and which in the set NB^-. If we put k in the set NB^+, the smallest possible coefficient in the cut for x_k is f_k. On the contrary, if we put it in the set NB^-, the smallest possible coefficient for x_k is $(\frac{f_j}{-1+f_j})(-1 + f_k)$. Therefore, k is put in NB^+ if

$$f_k \leq (\frac{f_j}{-1+f_j})(-1 + f_k)$$

equivalent to,

$$f_k(1 - f_j) \leq f_j(1 - f_k)$$

equivalent to,

$$f_k \leq f_j.$$

This choice of putting integer-constrained variables in the set NB^+ and NB^- yields the Gomory mixed integer cut mentioned at the beginning of this section.

A cutting plane algorithm tries to find the optimal solution of a given mixed integer program.

It does so by repeatedly solving the linear programming relaxation after the addition of each new cutting plane. Gomory [41] presented an appealing theoretical result by showing that the cutting plane algorithm, using his mixed integer cuts, can find an optimal solution in a finite number of steps if the objective function is integer valued. It still remains open whether this also holds for a real valued objective function [74].

Gomory cuts, although regarded since their introduction as theoretically significant, were not successful in practice until the 1990s [73, 63, 65, 64]. In 1996, Balas, Ceria, Cornuéjols, and Natraj showed the effectiveness of Gomory cuts as a computational tool by using them in a branch-and-cut framework. Many researchers have also tried to strengthen Gomory cuts so that comparing to the Gomory cut, the obtained new cut is further away from the basic feasible solution x^* of $\mathcal{L}(P)$, and thus cuts off more region from $\mathcal{L}(P)$ along with x^*. Anderson, Cornuéjols, and Li [5] and Balas and Perregaard [15] recently show two different ways of strengthening Gomory cuts.

2.1.7 Pseudo-Boolean Solvers

Integer programs, especially 0-1 integer programs, have received significant attention of the SAT community in last decade. This is mostly because of the introduction of some powerful SAT solvers [75, 57, 72, 60] and the existence of a reduction from a satisfiability instance to an equivalent 0-1 integer program. A 0-1 inequality is often referred as a pseudo-boolean constraint, and so a 0-1 integer program as a pseudo-boolean problem. Researchers have tried to use the powerful SAT solvers to solve the pseudo-boolean problems. Though a CNF clause is equivalent to a single pseudo-boolean constraint, a pseudo-boolean constraint may in some cases correspond to an exponential number of CNF clauses [71, 4]. Therefore, researchers have tried to implement the SAT solvers to handle pseudo-boolean constraints directly. This gives rise to a number of pseudo-boolean solvers [69, 3, 17, 4, 20, 66] for 0-1 integer programs. As a representative of state-of-the-art pseudo boolean solvers, we choose PBS Version 4.0 implemented by Aloul and Al-Rawi [2] to compare against other 0-1 integer program solvers. PBS 4.0 has become one of the best pseudo-boolean solver in the SAT solver competition held in SAT-2005, the eighth international conference on theory and applications of satisfiability testing. PBS4 is based on Zchaff2004 [56], an implementation of Chaff algorithm [60] for satisfiability instances, and the original PBS solver [3].

2.2 Heuristics for Integer Programs

Recall from Chapter 1 that we can categorize heuristics into two classes, namely feasibility heuristics and improvement heuristics. We review the main concepts and features of previously known heuristics in the next several sections.

2.2.1 Feasibility Heuristics

We consider the previously known heuristics, designed to find a feasible solution in their first phase, as the feasibility heuristics. Some of these heuristics have a second phase, namely improvement phase, which tries to improve the solution found in the first phase. We review existing feasibility heuristics in the next several sections.

2.2.2 Pivot and Complement and Its Successors

In 1980, Balas and Martin proposed the PIVOT AND COMPLEMENT (PC) [13] heuristic for 0-1 integer programs.

PC consists of two subroutines. The feasibility subroutine tries to find a feasible solution for the problem instance. The improvement subroutine applies a local search to improve the objective value of the solution found in the feasibility subroutine.

The feasibility subroutine is based on the fact that a feasible simplex tableau with all the 0-1 variables out of the basis gives a feasible solution for the integer program, since a 0-1 nonbasic variable has value 0 or value 1. With the aim of obtaining such a feasible simplex tableau, the heuristic starts with the optimal simplex tableau for the linear programming relaxation of the given 0-1 integer program and then performs a sequence of pivoting operations to make the 0-1 variables nonbasic. The feasibility subroutine consists of two phases. The search phase tries to minimize the measure of integer infeasibility by using two type of pivots. If this phase fails to find a feasible 0-1 solution, we say that the heuristic has reached a local minima. The restart phase tries to escape the local minima by using a third type of pivot and complementing one or more nonbasic 0-1 variables, where complementing a 0-1 variable means switching the value of the variable from v to $1 - v$. If the restart phase escapes the local minima, execution returns to the search phase.

The improvement subroutine tries to find a solution that yields a better objective value by complementing one or more 0-1 variables.

In 1986, Balas and Martin generalized PC to allow it to solve mixed integer programs. They

named this heuristic PIVOT AND SHIFT (PS) [14]. In PS, the major difference from PC is a replacement of the operation of complementing a 0-1 variable by the operation of shifting an integer-constrained variable by plus or minus one from the current value. In the corresponding new pivoting rule, integer-constrained variables replace 0-1 variables. The major drawback of both PC and PS is that they fail to produce a feasible solution for a large number of instances.

In 1994, Løkketangen, Jörnsten, and Storøy [55] applied a tabu search mechanism in the PC framework. They do not use the third type of pivot of PC when the search reaches a local minima; rather they use some tabu conditions to escape the local minima. This strategy can also search beyond the first feasible solution found with the motive of finding improved feasible solution. In the improvement phase, they consider complementing only one variable at a time by incorporating some tabu conditions to escape the local minima. Løkketangen et al. [55] showed that this tabu search based method performed better than PC. However, they tested their algorithm only on small multidimensional knapsack problems; as such it is not clear whether they will achieve similar performance on other classes of 0-1 integer programs.

In 2004, Balas, Schmieta, and Wallace [16] developed a new implementation of PIVOT AND SHIFT (PS[2004]) using the commercial linear programming and mixed integer programming solver Xpress Version-14.2. This heuristic starts with a sequence of pivoting with the same aims of the pivot operations of PC. When reaching a local minima or exceeding the time limit for the pivots, the heuristic defines a neighbourhood around the local minima and applies the Xpress mixed integer programming solver on the problem defining the neighbourhood. To define the aforementioned neighbourhood, they add the following constraints to the current problem.

$$\sum_{x_j \in S} \hat{x_j^*} - 1 \leq \sum_{x_j \in S} x_j \leq \sum_{x_j \in S} \hat{x_j^*} + 1,$$

where $S = \{x_i \in x_{\mathbb{Z}} \mid min\{x_i^* - \lfloor x_i^* \rfloor, \lceil x_i^* \rceil - x_i^*\} \leq \alpha\}$, x^* is the current basic solution of simplex tableau, and α is a small value, namely $\alpha = 0.2$. $\hat{x_j^*}$ denotes the nearest integer value of x_j^* and $x_{\mathbb{Z}}$ denotes the set of variables constrained to be integer in the given integer program.

If either the sequence of pivoting or the exploration of the aforementioned neighbourhood produces a solution x^*, execution switches to the improvement phase; otherwise, the heuristic calls the Xpress mixed integer programming solver to find a feasible solution and swithces to the improvement phase if Xpress finds a solution. If Xpress fails to find a solution, the heuristic aborts its execution. The improvement phase first tries to improve the found solution by a sequence of shifting on the nonbasic variables. When the alloted time for shifting elapses, the heuristic defines an improvement neighborhood and applies the Xpress mixed integer programming solver on the improvement neighbourhood. To define the improvement neighbourhood, they add the following

constraints to the current problem.

$$\sum_{x_j \in x_{\mathbb{Z}}} x_j^* - k \leq \sum_{x_j \in x_{\mathbb{Z}}} x_j \leq \sum_{x_j \in x_{\mathbb{Z}}} x_j^* + k,$$

where $k > 1$.

Balas et al. compared this heuristic against the Xpress mixed integer programming solver and showed that the heuristic performed better comparing to Xpress on the benchmark instances. They did not show any comparison of the heuristic against the other existing heuristics.

2.2.3 Octahedral Neighbourhood Enumeration

In 2001, Balas et al. [12] presented the heuristic Octahedral Neighborhood Enumeration (OCTANE) for solving 0-1 integer programs. This heuristic defines an integer neighborhood of the fractional solution to the linear programming relaxation of the integer program and searches that neighborhood in a particular direction. A unit hypercube K centered at the origin defines this integer neighborhood, where K is expressed by $\{x \in \mathbb{R}^n \mid -e/2 \leq x_i \leq e/2, e = (1, 1, \ldots, 1)\}$. Balas et al. define an octahedron K^* circumscribing K by $\{x \in \mathbb{R}^n \mid \delta x \leq n/2, \forall \delta \in \{\pm 1\}^n\}$. Figure 2.6 shows K and K^* in two-dimensional space. For each facet δ of $(K^* + e/2)$, there is a corresponding vertex x of $(K + e/2)$ defined by $x = \delta/2 + e/2$.

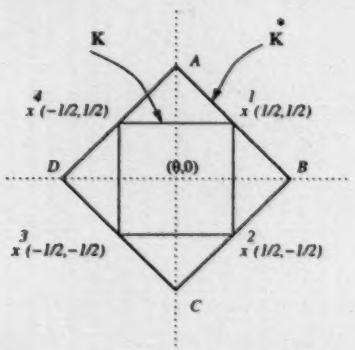

Figure 2.6: Illustration of K and K^* in two-dimension. The thin and thick edges represent the edges of K and K^* respectively.

The basic steps of the heuristic are the following.

Step 1: Construct a directional ray, specified by a direction vector, originating from an linear programming relaxation optimal solution.

Step 2: Determine the first k intersections of the directional ray with the facets of octahedron $(K^* + e/2)$. Determine the 0-1 points on the hypercube $(K + e/2)$ corresponding to the first k intersected facets of $(K^* + e/2)$.

Step 3: Consider each 0-1 solution as the candidate solution and determine the feasible ones for the 0-1 integer program.

The heuristic presents a systematic technique to enumerate the first k facets of $(K^* + e/2)$ intersected by the directional ray. Checking all k candidate solutions for feasibility in Step 3 is computationally inefficient when the size of the given problem is large. To eliminate this drawback, the heuristic shows a modified technique that tries to avoid enumerating facets that correspond to infeasible 0-1 points. For this purpose, it uses canonical inequalities which have the form $z_0 \leq sx \leq s_0$ where s is in $\{0, \pm 1\}^n$ and s_0 and z_0 are integers. The modified technique then enumerates those facets which intersect the directional ray and whose corresponding 0-1 points are satisfied by the canonical inequality.

Balas et al. applied this heuristic at different nodes of the branch-and-cut framework so that the directional ray of search could originate from different points. They also tried a variety of directions for enumeration. The average ray (the average of the extreme rays of the cone at the current linear programming optimal solution) and the objective ray (the normal vector of the objective ray directed inward) were two such chosen directions. The empirical results reported in [12] suggest that OCTANE is a competitive alternative of PS.

2.2.4 Pivot, Cut, and Dive

In 2001, Nediak and Eckstein [61] presented PIVOT, CUT, AND DIVE (PCD), a 0-1 mixed integer program heuristic. As in PC, the main idea of this heuristic is to perform a sequence of simplex tableau pivots to find a feasible solution for the 0-1 mixed integer program. But unlike in PC, a concave merit function determines the pivoting nonbasic variable. The merit function is designed in such a way that it evaluates to zero at all integer feasible points and to some positive value for integer infeasible points. It thus gives a measure of integer infeasibility of the current solution. The merit function used is

$$\psi(x) = \sum_{x_i \in x_{01}} \phi(x_i)$$

where x_{01} is the set of variables constrained to be 0 or 1, and

$$\phi(x_i) = 1 - \begin{cases} (\frac{x_i - \alpha}{\alpha})^2, & x_i \leq \alpha \\ (\frac{x_i - \alpha}{1 - \alpha})^2, & x_i \geq \alpha \end{cases}$$

where $\alpha \in (0,1)$.

The heuristic first defines the following two pivot selection rules based on an approximation of the merit function defined by the local gradient of the merit function.

Type 1: A pivot that improves the approximation of the merit function maximum and does not make the objective value worse.

Type 2: A pivot that improves the approximation of the merit function with the minimum worsening of the objective value.

If no pivot of Type 1 or 2 is found, the heuristic checks all possible pivoting options and looks for one that improves the original merit function with an allowable limit of worsening the objective value. This is the Type 3 pivot, called a probing pivot.

If at any step of pivoting the heuristic does not find a pivot of Type 1, 2 or 3, it considers to include a convexity cut or to branch using a vertex cut.

If $\alpha x \geq \beta$ is a cutting plane, its distance from the current solution x^* is $\frac{\beta - \alpha x^*}{\|\alpha\|}$.

The convexity cut only cuts off the current fractional solution; it does not cut off any integer feasible solution. If the convexity cut has a distance greater than the vertex cut from the current fractional solution, then the convexity cut is added to the problem. The resulting problem is re-optimized, namely its linear programming relaxation is solved, and execution returns to the pivoting phase.

The vertex cut cuts off the current fractional solution as well as a set of possible 0-1 feasible solutions with $x_i = 0, \forall x_i \in Q_0$ and $x_i = 1, \forall x_i \in Q_1$; we will see the definition of Q_0 and Q_1 in § 2.2.6. Therefore, if the vertex cut has a distance greater than the convexity cut from the current fractional solution, then the heuristic generates two subproblems. First subproblem is the one that includes $x_i = 0, \forall x_i \in Q_0$ and $x_i = 1, \forall x_i \in Q_1$, and the second subproblem is the one that includes the vertex cut. Then the heuristic branches to the first subproblem. If the first branch has no feasible solution, then the heuristic branches to the second subproblem.

After branching to the new subproblem, the heuristic re-optimizes the subproblem and again transfers execution to the pivoting phase with that subproblem.

Nediak and Eckstein experimented with this heuristic on the 49 0-1 mixed integer program instances of MIPLIB 3.0. Out of the 49 instances, they excluded 7 instances because the linear program solver they used could not handle those instances. Out of the 42 instances, PCD failed in 10 instances. They did not compare their heuristic with any other heuristics or the commercial solvers.

Following the descriptions of the convexity cut and the vertex cut by Nediak and Eckstein in

PCD [61], we present brief descriptions of convexity cut and vertex cut in § 2.2.5 and § 2.2.6.

2.2.5 Convexity Cut

In [7] Balas introduced the convexity cut in integer programming. Let x^* be the current fractional solution to the linear programming relaxation of a given integer program. The edges $x(i) = x^* - C_i v_i$ for $v_i \geq 0$, $\forall x_i \in x_N$ define the polyhedral cone formed at this vertex, where C_i is the current column in the simplex tableau for the nonbasic variable x_i, and v_i is a parameter illustrating how much change has been made to the value of x_i from its current lower or upper bound value at x^*. The simplex tableau only provides the values C_{ij} of the vector C_i corresponding to the basic variables x_j. All other values of C_{ij} are 0 except for the value of C_{ii}. The value of C_{ii} is plus or minus one depending on the value of x_i at its upper bound or lower bound.

Assume that v_i^* is the value of v_i for each x_i in x_N that makes $\psi(x(i)) = 0$. Then the hyperplane passing through the point identified by v_i^* is $\sum_{x_i \in x_N} (\frac{1}{v_i^*}) v_i = 1$. Therefore, the equation of the cutting plane is

$$\sum_{x_i \in x_N} \left(\frac{1}{v_i^*}\right) v_i \geq 1,$$

where v_i is x_i (or $u_i - x_i$) if x_i is at its lower (upper) bound in x^*. u_i represents the upper bound of the variable x_i.

2.2.6 Vertex Cut

These cuts have the form $\sum_{x_i \in Q_0} x_i + \sum_{x_i \in Q_1} (1 - x_i) \geq 1$, where Q_0 is a subset of V_0, and Q_1 is a subset of V_1, with V_0 and V_1 being the sets of the binary variables whose current value's nearest integers are 0 and 1 respectively. Choosing different set for Q_0 and Q_1, one can find different vertex cuts having different distances from the current solution. In PCD, Nediak and Eckstein generated vertex cuts so that that each cut's distance from the current fractional solution was bounded.

2.2.7 Tabu Search in Solving Integer Programs

Recall that in 1994, Løkketangen, Jörnsten, and Storøy [55] incorporated a tabu search mechanism into the PC framework. In 1998, Løkketangen and Glover [54] gave a more general application of tabu search for solving 0-1 mixed integer programs.

Since an extreme point of the underlying feasible region corresponding to a linear programming relaxation may be the feasible solution of a 0-1 mixed integer program, Løkketangen and Glover apply tabu search with the aim of visiting such extreme points.

Løkketangen and Glover select moves, namely pivots, for the search from a possible set of moves based on the measure of integer infeasibility and objective value. A main feature of tabu search is to set up some condition to make a set of moves prohibited or tabu. In this particular heuristic, a variable becomes tabu whenever it becomes nonbasic from basic and remains tabu for a specified time. Another feature of tabu search is to set up an aspiration condition, a condition that makes a tabu move acceptable if it is satisfied. The aspiration condition to accept a tabu move in [54] is to have a new basic feasible solution of the linear programming relaxation that has the measure of integer infeasibility within a certain limit.

Løkketangen and Glover also experimented with a probabilistic approach to select a move instead of relying on tabu conditions. They tested their algorithms only on small multidimensional knapsack problems; as such it is not clear whether they will achieve similar performance on other classes of integer programs.

2.2.8 Heuristics Based on Interior Path

In 1969, Hillier [43] proposed a heuristic for solving general integer programs using the so-called interior paths. The idea is to select two points. One is the linear programming relaxation optimal solution x_1, an extreme point of the linear programming relaxation feasible region. The other is a point x_2 such that the path from x_1 to x_2 goes through the interior of the feasible region. Hillier showed two ways to select a point x_2. The method then moves from x_1 to x_2 using a parameter and checks whether the rounding of the points in this path gives an integer feasible solution. If they do not, it searches the neighborhood of those points by shifting the values of integer variables by plus or minus one. The interior path is defined by $x = x_1 + \alpha(x_2 - x_1)$, where α is the parameter to move along the path.

In 1974, Ibaraki, Ohashi, and Mine [45] and in 1979, Faaland and Hillier [33] proposed similar heuristics for solving general mixed integer programs. One difference in their method from that of Hillier [43] is to construct the interior path as a set of piecewise linear segments. This method determines a set of points $\{x_1, x_2, \ldots, x_k\}$ where x_1 is the optimal solution to the linear programming relaxation of the integer program, and $\{x_2, \ldots, x_k\}$ lie in the interior of polyhedron. Then $\{x_1x_2, x_2x_k, \ldots, x_{k-1}x_k\}$ is the set of line segments. The method starts searching feasible so-

lutions for the mixed integer program from x_1. During the traversal the method checks whether rounding at a non-integer point gives a solution; if not, it searches the neighborhood of that point by shifting the values of integer variables by plus or minus one.

All these methods go through an improvement phase to improve the found feasible solution by shifting the values of integer variables by plus or minus one.

2.2.9 Feasibility Pump

In 2005, Fischetti, Glover, and Lodi [34] introduce FEASIBILITY PUMP (FP), a mixed integer programming heuristic. This is the most recent among all the existing feasibility heuristics.

FP starts with the linear programming relaxation solution and, as long as it does not find a solution, continues to solve a sequence of linear programs guided by the rounding of the previous step's linear programming relaxation solution; random perturbations are introduced in case of cycling. More explicitly, suppose the integer program P given by $min \ \{c^T x \mid Ax \geq b, \ x_j \in \{0,1\} \ \forall j \in I\}$ is the input to the FP. Further assume that x^* is an optimal solution to the linear programming relaxation $\mathcal{L}(P)$ of P, and $\tilde{x}$ is the rounding of x^*. FP defines a distance function

$$\Delta(x, \tilde{x}) := \sum_{j \in I \wedge \tilde{x}_j = 0} x_j + \sum_{j \in I \wedge \tilde{x}_j = 1} (1 - x_j).$$

If $\Delta(x^*, \tilde{x}) = 0$, x^* is a feasible solution of P. If not, FP solves the linear program, $min \ \{\Delta(x, \tilde{x}) \mid Ax \geq b, \ 0 \leq x_j \leq 1 \ \forall j \in I\}$, which gives the closest point x^*_{new} of $\tilde{x}$ on the polyhedra associated with $\mathcal{L}(P)$. FP considers x^*_{new} as the x^* of the next iteration and continues as mentioned earlier. FP perturbs $\tilde{x}$ in some random way if a cycle is detected in last three iterations and after certain number of iterations.

Fischetti et al. implemented FP using the commercial linear programming solver Cplex. They compared FP against the commercial mixed integer programming solver Cplex Version 8.1 and showed that FP performed better comparing to the Cplex on the benchmark instances. They did not show any comparison of FP against the other existing heuristics.

2.2.10 Improvement Heuristics

We consider the previously known heuristics, designed only to find improved solutions from a known solution, as the improvement heuristics. The existing such heuristics apply themselves at different

nodes of the mixed integer program search tree generated by either a branch-and-bound or a branch-and-cut solver. We review them in § 2.2.11 and § 2.2.12.

2.2.11 Local Branching

LB, designed by Fischetti and Lodi [35], is the first strategy that introduces the idea of exploring a promising neighbourhood around a known mixed integer program solution by defining the neighbourhood as another mixed integer program instance and exploring it using a black-box mixed integer program solver, namely either a branch-and-bound or a branch-and-cut solver.

LB defines the neighbourhood of a feasible solution x^* by limiting at some integer p the number of 0-1 variables currently at 0 or 1 that can switch their bounds. This is often called *soft fixing*.

LB achieves this by adding to the instance the inequality $D(x, x^*) \leq p$, where

$$D(x, x^*) := \sum_{j \in V_0} x_j + \sum_{j \in V_1} (1 - x_j),$$

with V_0 and V_1 being the index sets of the 0-1 variables that are at 0 and 1 respectively in x^*.

We find two different implementations of LB. Originally, Fischetti and Lodi [35] implemented LB as an heuristic as well as an external branching framework, which creates branches in the search tree by $D(x, x^*) \leq p$ and $D(x, x^*) \geq p + 1$ as opposed to the standard branching on the variables in the branch-and-bound framework. They obtained the diversification, which switches the search in a different region of the MIP feasible space, by defining the neighbourhoods with a change in the value of the parameter p. Later, Danna, Rothberg, and Pape [29] implemented LB solely as a heuristic, and obtained the diversification by defining the neighbourhoods on the new solutions found during the mixed integer program search tree exploration. Danna et.al. also showed that their implementation of LB outperformed the original.

Figure 2.7 describes the operation sequence of LB at a particular node of the mixed integer program search tree. At the termination of the procedure, the algorithm resumes the exploration of the mixed integer program search tree, and if the procedure finds a new mixed integer program solution, the algorithm updates the mixed integer program solution at the mixed integer program search tree. The algorithm first calls the LB procedure at the node where the mixed integer program search tree finds its first mixed integer program solution, and then, as a process of diversification, every time the mixed integer program search tree finds a new mixed integer program solution.

Fischetti and Lodi [35] showed that it is possible to define $D(x, x^*)$ to handle general integer

variables, but doing so requires the introduction of a new set of variables. $D(x, x^*)$ used in both [35] and [29] is defined only on 0-1 variables.

Procedure LB_at_tree_node
INPUT: a 0-1 mixed integer problem P, a current MIP solution x_o^*,
parameter p, and a node limit nl.
OUTPUT: A new MIP solution x^* (x_o^* in case of failure in finding a new solution).

1. $x^* \leftarrow x_o^*$, $p_{current} \leftarrow p$, exploredAndNoSolution $\leftarrow$false
2. repeat
3. compute the LB inequality $D(x, x^*) \leq p_{current}$
4. construct $P+$ from P by adding the LB inequality
5. Apply black-box MIP solver to $P+$ with node limit nl and
an objective cutoff equal to the objective value provided by x^*
6. if (a new solution x_{new} is obtained) then
7. $x^* \leftarrow x_{new}$, $p_{current} \leftarrow p$
8. else if (node limit reached without having a new solution) then
9. $p_{current} \leftarrow p_{current}/2$
10. else exploredAndNoSolution $\leftarrow$true
11. until ($p_{current} < 5$ or exploredAndNoSolution)
12. return x^*

Figure 2.7: A pseudo-code version of the algortihm LB due to Danna et al. [29].

2.2.12 Relaxation Induced Neighbourhood Search

During the exploration of the mixed integer program search tree, the relaxation solutions at those successive nodes that are not pruned always provide a better objective value than that of the current mixed integer program solution. Using this idea, Danna, Rothberg, and Pape [29] introduce RINS using the intuition that, in improved mixed integer program solutions, it is more likely for those variables, that agree in the current mixed integer program solution and current node relaxation solution, to stay at the same values. Thus RINS defines the promising neighbourhood by fixing all variables whose values at the current mixed integer program solution are equal to their respective values at the current node relaxation solution. This is often called *hard fixing*.

Figure 2.8 describes the operation sequence of RINS at a particular node of the mixed integer program search tree. At the termination of the procedure, the algorithm resumes the exploration of the mixed integer program search tree. If the procedure finds a new mixed integer program solution, the algorithm updates the mixed integer program solution at the mixed integer program search tree.

As noted in [29], consecutive nodes of the mixed integer program search tree provide almost identical relaxation solutions. Therefore, the algorithm calls RINS procedure at every f nodes for

some reasonably large f.

Danna et al. compared RINS against the commercial mixed integer programming solver Cplex and the heuristic LB. They showed that RINS performed better than both the Cplex and the LB.

Procedure RINS_at_tree_node
INPUT: a 0-1 mixed integer problem P, the current MIP solution x_o^*,
the current node relaxation solution x_{node}, and a node limit nl.
OUTPUT: A new MIP solution x^* (x_o^* in case of failure in finding a new solution).

1. $x^* \leftarrow x_o^*$
2. construct $P+$ from P by fixing the variables that agree in x_o^* and x_{node}
3. Apply black-box MIP solver to $P+$ with node limit nl and
   an objective cutoff equal to the objective value provided by x^*
4. if (a new solution x_{new} is obtained) then
5. $x^* \leftarrow x_{new}$
6. return x^*

Figure 2.8: A pseudo-code version of the algorithm RINS presented by Danna et al. [29])

2.3 Benchmark Integer Program Instances

2.3.1 Existing Library of Integer Program Instances

To measure the performance of mixed integer program solvers, it is useful to have a set of test instances from different areas of optimization. Prior to 1998, mixed integer program solvers were usually experimented on using randomly generated multidimensional knapsack instances. In 1992, Bixby, Boyd and Indovina [18] created MIPLIB, the first electronic library of real world mixed integer programs, to meet the requirements of researcher. In 1998, Bixby, Ceria, McZeal and Savelsbergh [19] updated the MIPLIB. Since then, MIPLIB has become a standard test suite used to compare the performance of mixed integer program solvers. In 2003, the latest update is made to the MIPLIB by Martin, Achterberg and Koch [58]. Table B.1 of Appendix B describes all the mixed integer program instances of MIPLIB 2003.

In addition to the instances available from MIPLIB, recent research [29, 34, 35] on heuristics for mixed integer program uses the mixed integer program instances available from DEIS operations research group electronic library [31]. Table B.2 of Appendix B describes all the mixed integer program instances from DEIS electronic library.

Recent heuristics [34, 29] on mixed integer programs also use five job-shop scheduling instances

with earliness and tardiness costs. The heuristic in [29] uses eleven network design and multi-commodity routing instances. Table B.3 of Appendix B describes these sixteen instances.

In our experiments, we used all the mixed integer program instances noted above. This set of instances represents the current hard mixed integer program instances available from the community and acts as the benchmarks in recent heuristic research for mixed integer programs.

2.3.2 Generation of Hard Integer Program Instances

Most of the integer program instances in the benchmark test suite come either from the instances generated randomly for various optimization problems using their most common integer program model or from the real-world instances of different optimization problems; they are hard to solve by the existing solvers.

The market-sharing problem introduced by Williams [73] is one such optimization problem. The problem as described in [73] is as follows:

> *A large company has two divisions D_1 and D_2. The company supplies retailers with several products. The goal is to allocate each retailer to either division D_1 or division D_2 so that D_1 controls 40% of the company's market for each product and D_2 the remaining 60% or, if such a perfect 40/60 split is not possible for all the products, to minimize the sum of percentage deviations from the 40/60 split.*

A natural integer program formulation for this problem is given by Cornuéjols and Dawande [27].

$$\begin{aligned} \min \quad & \textstyle\sum_{i=1}^{m} |s_i| \\ \text{s.t.} \quad & \textstyle\sum_{j=1}^{n} a_{ij} x_j + s_i = b_i \qquad i = 1, ..., m \\ & x_j \in \{0, 1\} \qquad j = 1, ..., n \\ & s_i \quad free \qquad i = 1, ..., m, \end{aligned}$$

where n, m, a_{ij} are the number of retailers, the number of products, and the demand of retailer j for product i respectively. One can determine b_i from the desired market split between the two divisions D_1 and D_2.

Cornuéjols and Dawande [27] showed that choosing each integer a_{ij} uniformly between 0 and 99, setting $n = 10(m - 1)$, and asking for a 50/50 split yields a class of optimality-hard 0-1 program instances for the existing integer program solvers, where by 'optimality-hard' we mean finding the optimal solution with the certificate of optimality is hard. For the 50/50 split, b_i is

equal to $\lfloor 0.5 \times \sum_{j=1}^{n} a_{ij} \rfloor$ for $1 \leq i \leq m$. We refer to this formulation of the Williams's market-sharing problem defined by these choices as the Cornuéjols-Dawande optimality model, and the instances generated following this model as the Cornuéjols-Dawande optimality-hard instances. In [27] Cornuéjols and Dawande pointed that their optimality-hard instances are related to the knapsack instances considered by Chvátal [22], since their optimality-hard instances are similar to the instances of multi-dimensional knapsack problems. Chvátal considered the following knapsack problem,

$$\begin{aligned} \max \quad & \textstyle\sum_{j=1}^{n} a_j x_j \\ \text{s.t.} \quad & \textstyle\sum_{j=1}^{n} a_j x_j \leq \lfloor \sum_{j=1}^{n} a_j / 2 \rfloor \\ & x_j \in \{0,1\} \quad j = 1, ..., n, \end{aligned}$$

and showed that choosing each integer a_j uniformly between 1 and 10^3 yields a class of hard knapsack instances for branch-and-bound solvers.

Notice that the Cornuéjols-Dawande optimality model is not suitable for generating feasibility-hard instances, since any choice of x satisfying $x_j \in \{0,1\}$ for all j yields a feasible solution for instances of this model. However, Cornuéjols and Dawande showed the following feasibility formulation corresponding to their optimality model.

$$\begin{aligned} & x \\ \text{s.t.} \quad & \textstyle\sum_{j=1}^{n} a_{ij} x_j = b_i \qquad i = 1, ..., m \\ & x_j \in \{0,1\} \qquad j = 1, ..., n \end{aligned}$$

Notice that the constraints of the optimality model drop their slack/surplus variables to become the constraints of this feasibility formulation. This feasibility problem is NP-complete [27], and a feasible solution to an instance of this formulation exists only if the optimal objective value of the corresponding optimality-hard instance is zero. We refer to this feasibility formulation of the Williams's market-sharing problem as the Cornuéjols-Dawande feasibility model, and the instances generated following this model as the Cornuéjols-Dawande feasibility-hard instances.

The pseudo-randomly generated Cornuéjols-Dawande optimality-hard instances have the optimal objective value larger than zero for most of the instances. Therefore, the Cornuéjols-Dawande feasibility-hard instances corresponding to these generated optimality-hard instances have no feasible solution. Aardal, Bixby, Hurkens, Lenstra, and Smeltink [1] analyzed the expected number of solutions for the Cornuéjols-Dawande feasibility-hard instances with different n and m. Following

their analysis, it is possible to choose n and m for which the pseudo-randomly generated Cornuéjols-Dawande feasibility-hard instances are with high probability feasible. We pseudo-randomly generated some Cornuéjols-Dawande feasibility-hard instances to experiment with different feasibility heuristics, and some Cornuéjols-Dawande optimality-hard instances to experiment with different improvement heuristics.

In this thesis, we also present a new class of hard 0-1 program instances based on a modified form of Williams's market-sharing problem. We refer the instances generated according to the modified form of Williams's market-sharing problem as the constrained market-sharing instances. We pseudo-randomly generated some constrained market-sharing instances to experiment with different heuristics.

Chapter 3

Pivot and Gomory Cut

In this chapter we present PIVOT AND GOMORY CUT (PGC), a new mixed integer programming feasibility heuristic. Since PGC uses pivoting rules from PC, and since we implemented a version of PC to test against PGC, we begin our description of PGC by first discussing PC.

3.1 Pivot and Complement

In 1980, Balas and Martin introduced PC. Although introduced originally as a 0-1 integer programming heuristic, the pivoting rules of PC generalize naturally to the mixed integer programming setting, as mentioned in [14]. Since recent feasibility heuristics have been presented for 0-1 mixed integer programs, rather than just 0-1 integer programs, we implemented a 0-1 mixed integer version of PC. We now describe our implementation of this version of PC.

In § 2.1 we gave the formulation of an integer program. We now give the analogous formulation of a 0-1 mixed integer program. We assume that the input program P has the form shown below, where c, x, b, A have dimensions $n, n, m, m \times n$ respectively, $N = \{1, \ldots, n\}$ is the set of variable indices of P, I is the subset of N indexing the binary variables of P, and, for each index j for which x_j is a non-binary variable, l_j and u_j denote the respective lower and upper bounds for x_j.

PC transforms P into the equivalent program P^+ by adding the m-dimensional vector y. The scalar elements of x and y are the *decision variables* and *surplus variables* respectively. The non-binary decision variables and the surplus variables are the *continuous variables*. $\mathcal{L}(P^+)$ is the linear program obtained from P^+ by relaxing the integrality constraints on the binary decision variables.

Thus we have

$$
\begin{aligned}
P: &\quad \min \{\, cx \mid Ax \geq b,\ x_i \in \{0,1\}\ \forall i \in I,\ l_j \leq x_j \leq u_j\ \forall j \in N - I \} \\
P^+: &\quad \min \{\, cx \mid Ax - y = b,\ x_i \in \{0,1\}\ \forall i \in I,\ l_j \leq x_j \leq u_j\ \forall j \in N - I,\ y \geq 0 \} \\
\mathcal{L}(P^+): &\quad \min \{\, cx \mid Ax - y = b,\ 0 \leq x_i \leq 1\ \forall i \in I,\ l_j \leq x_j \leq u_j\ \forall j \in N - I,\ y \geq 0 \}\,.
\end{aligned}
$$

PC is based on a sequence of pivot operations that exchange a nonbasic variable with a basic variable in a simplex tableau associated with $\mathcal{L}(P^+)$. The first step is to use the bounded variable revised simplex algorithm to look for an optimal solution to $\mathcal{L}(P^+)$. If $\mathcal{L}(P^+)$ has no such solution, then PC reports that P has no feasible solution; otherwise, execution transfers to the feasibility subroutine with an $\mathcal{L}(P^+)$-optimal basic feasible solution x^*.

The PC feasibility subroutine has two phases, search and restart. In the search phase, PC tries to decrease the extent to which x^* is P-infeasible by repeated tableau pivoting, as follows. Each binary decision variable of x^* which is not in the basis will have its value equal to either its upper or lower bound and therefore be integral. Thus, the primary P-infeasibility measure is the number of basic binary decision variables of x^*, and the search phase applies the following pivot whenever possible.

> PC *Type 1 Pivot*: The pivot that exchanges a basic binary decision variable with a nonbasic continuous variable, maintains $\mathcal{L}(P^+)$-feasibility, and, among all such pivots, minimizes the objective functions.

Once no more such pivots are available, PC considers a secondary P-infeasibility measure defined by $I_P(x^*) = \sum_{j \in I} \min\{x_j^*, 1 - x_j^*\}$.

> PC *Type 2 Pivot*: The first pivot found that exchanges a basic continuous variable with a nonbasic continuous variable, or a basic binary decision variable with a nonbasic binary decision variable, maintains $\mathcal{L}(P^+)$-feasibility, and reduces $I_P(x^*)$.

Pivoting continues until either PC finds a P-feasible solution or reaches a dead end, namely there are no available Type 1 or Type 2 pivots. When PC encounters such a dead end, it checks whether rounding or truncating the binary decision variables of x^* yields a P-feasible solution. If so, then PC terminates successfully. Otherwise, the restart phase begins.

In the restart phase, PC perturbs the dead end by a third kind of pivot into an $\mathcal{L}(P^+)$-infeasible solution. The measure of infeasibility used is $I_L(x^*) = \sum_{j \in I} \max\{0, -x_j^*, x_j^* - 1\} + \sum_{j \in N-I} \max\{0, l_j - x_j^*, x_j^* - u_j\} + \sum_{y_j \in y} \max\{0, -y_j^*\}$.

> PC *Type 3 Pivot*: The pivot that exchanges a basic binary decision variable with a non-basic continuous variable and leaves the entering continuous variable positive, and, among all such pivots, minimizes $I_L(x^*)$.

After such a pivot, x^* is a basic infeasible solution of $\mathcal{L}(P^+)$. The restart phase next attempts to bring $I_L(x^*)$ to zero by repeatedly complementing one or two nonbasic binary decision variables, namely by changing their values from one bound to the other. If PC finds an $\mathcal{L}(P^+)$-feasible solution by this complementation process, it first rounds and then truncates the binary decision variables of x^*. If either process yields a P-feasible solution, then PC terminates successfully; otherwise, execution transfers back to the search phase. If PC does not find an $\mathcal{L}(P^+)$-feasible solution by the repeated complementation of any single or pair of nonbasic binary decision variables, it aborts.

This completes the description of our implementation of PC. The original versions of PC [13, 14] also have a complementation-based improvement subroutine to improve a P-feasible solution found by the feasibility subroutine. We omit any further discussion of this subroutine here, as we do not include it in PGC, where our goal is only to find a feasible solution.

We now give a step-by-step example that illustrates the execution of PC. Consider the following as the given integer program P.

$$
\begin{aligned}
\min \quad & 5x_1 + 6x_2 + 9x_3 - 5x_4 - 3x_5 \\
\text{s.t.} \quad & 7x_1 + 9x_2 + 9x_3 + x_4 + 5x_5 \geq 15 \\
& 3x_1 + 6x_2 + 7x_3 + 3x_5 \geq 9 \\
& 5x_1 + x_2 + x_3 + 6x_4 + 5x_5 \leq 9 \\
& 8x_2 + 6x_3 + 6x_5 \leq 10 \\
& x_1, x_2, x_3, x_4, x_5 \in \{0, 1\}.
\end{aligned}
$$

First, PC transforms P to P^+ by adding the slack/surplus variables as follows.

$$\begin{aligned}
\min \quad & 5x_1 + 6x_2 + 9x_3 - 5x_4 - 3x_5 \\
\text{s.t.} \quad & 7x_1 + 9x_2 + 9x_3 + x_4 + 5x_5 - x_6 = 15 \\
& 3x_1 + 6x_2 + 7x_3 + 3x_5 - x_7 = 9 \\
& 5x_1 + x_2 + x_3 + 6x_4 + 5x_5 + x_8 = 9 \\
& 8x_2 + 6x_3 + 6x_5 + x_9 = 10 \\
& x_1, x_2, x_3, x_4, x_5 \in \{0,1\}, x_6, x_7, x_8, x_9 \geq 0.
\end{aligned}$$

Next, PC solves the linear programming relaxation $\mathcal{L}(P^+)$ of P^+ using the bounded variable revised simplex algorithm. This gives the following optimal simplex tableau. In the tableau, C denotes the objective row, x_1, x_3, x_4, and x_5 are the basic variables and the remaining are the non-basic variables.

$$\begin{aligned}
5 + x_6 + x_7 + x_8 + x_9 &= C \\
1.1978 - 1.0989x_2 + 0.2812x_6 - 0.5781x_7 + 0.0468x_8 - 0.2604x_9 &= x_5 \\
-0.1666 + 1.0833x_2 - 0.5x_6 + 0.75x_7 - 0.25x_8 + 0.1666x_9 &= x_4 \\
0.7083 - 0.3541x_2 + 0.375x_6 - 0.4375x_7 + 0.0625x_8 + 0.0416x_9 &= x_1 \\
0.4687 - 0.2343x_2 - 0.2812x_6 + 0.5781x_7 - 0.0468x_8 + 0.0937x_9 &= x_3
\end{aligned}$$

Since the solution $(x_1, x_2, x_3, x_4, x_5) = (0.3541, 1.0, 0.2344, 0.9167, 0.0989)$ is not P-feasible, PC enters the search phase of feasibility subroutine with this optimal simplex tableau. There, it finds a desired Type 1 pivot, namely exchanging x_7 and x_4. Performing this pivot yields the following tableau with x_1, x_3, x_5, and x_7 as the basic variables.

$$\begin{aligned}
5.2222 - 1.4444x_2 + 1.6666x_6 + 1.3333x_4 + 1.3333x_8 + 0.7777x_9 &= C \\
1.0693 - 0.2638x_2 - 0.1041x_6 - 0.7708x_4 - 0.1458x_8 - 0.1319x_9 &= x_5 \\
0.2222 - 1.4444x_2 + 0.6666x_6 + 1.3333x_4 + 0.3333x_8 - 0.2222x_9 &= x_7 \\
0.6112 + 0.2777x_2 + 0.0833x_6 - 0.5833x_4 - 0.0833x_8 + 0.1388x_9 &= x_1 \\
0.5972 - 1.0694x_2 + 0.1041x_6 + 0.7708x_4 + 0.1458x_8 - 0.0347x_9 &= x_3
\end{aligned}$$

Since the solution $(x_1, \cdots, x_5) = (0.3055, 1.0, 0.2986, 1.0, 0.0347)$ is not yet P-feasible, it again looks for a Type 1 pivot, and finds one between x_9 and x_5. Performing this pivot yields the following tableau with x_1, x_3, x_7, and x_9 as the basic variables.

$$11.5262 - 3x_2 + 1.0526x_6 - 3.2105x_4 + 0.4736x_8 - 5.8947x_5 = C$$

$$8.1053 - 2x_2 - 0.7894x_6 - 5.8421x_4 - 1.1052x_8 - 7.5789x_5 = x_9$$

$$-1.5789 - x_2 + 0.8421x_6 + 2.6315x_4 + 0.5789x_8 + 1.6842x_5 = x_7$$

$$1.7368 - 0.0263x_6 - 1.3947x_4 - 0.2368x_8 - 1.0526x_5 = x_1$$

$$0.3158 - x_2 + 0.1315x_6 + 0.9736x_4 + 0.1842x_8 + 0.2631x_5 = x_3$$

Since the solution $(x_1, \cdots, x_5)$ = (0.3421, 1.0, 0.2894, 1.0, 0.0) is not yet P-feasible, it again looks for a Type 1 pivot. This time there is no such pivot, so it next looks for a Type 2 pivot. It finds one between x_8 and x_9. Performing this pivot yields the following tableau with x_1, x_3, x_7, and x_8 as the basic variables.

$$14.9998 - 3.8571x_2 + 0.7142x_6 - 5.7142x_4 - 0.4285x_9 - 9.1428x_5 = C$$

$$7.3337 - 1.8095x_2 - 0.7142x_6 - 5.2857x_4 - 0.9047x_9 - 6.8571x_5 = x_8$$

$$2.6666 - 2.0476x_2 + 0.4285x_6 - 0.4285x_4 - 0.5238x_9 - 2.2857x_5 = x_7$$

$$0.4285x_2 + 0.1428x_6 - 0.1428x_4 + 0.2142x_9 + 0.5714x_5 = x_1$$

$$1.6666 - 1.3333x_2 - 0.1666x_9 - x_5 = x3$$

The solution $(x_1, \cdots, x_5)$ = (0.2857, 1.0, 0.3333, 1.0, 0.0) reduces $I_p(x^*)$ from 0.3421 + 0 + 0.2894 + 0 + 0 = 0.6315 to 0.2857+0+0.3333+0+0 = 0.6190, but is not yet P-feasible. PC again looks for a Type 1 pivot. No such pivot is found, so it next looks for a Type 2 pivot. It finds one between x_2 and x_3. Performing this pivot yields the following tableau with x_1, x_2, x_7, and x_8 as the basic variables.

$$10.1785 + 2.8928x_3 + 0.7142x_6 - 5.7142x_4 + 0.0535x_9 - 6.25x_5 = C$$

$$5.0718 + 1.3571x_3 - 0.7142x_6 - 5.2857x_4 - 0.6785x_9 - 5.5x_5 = x_8$$

$$0.107 + 1.5357x_3 + 0.4285x_6 - 0.4285x_4 - 0.2678x_9 - 0.75x_5 = x_7$$

$$0.5356 - 0.3214x_3 + 0.1428x_6 - 0.1428x_4 + 0.1607x_9 + 0.25x_5 = x_1$$

$$1.25 - 0.75x_3 - 0.125x_9 - 0.75x_5 = x_2$$

The solution $(x_1, \cdots, x_5)$ = (0.0714, 0.5, 1.0, 1.0, 0.0) reduces the $I_p(x^*)$ to 0.0714+0.5+0+0+0 = 0.5714, but is not yet P-feasible. PC again looks for a Type 1 pivot. No such pivot is found, so it next looks for a Type 2 pivot. Since there is no Type 2 pivot, it checks whether either rounding or truncating of solution gives P-feasible solution. Since it does not, PC is at a dead end and so next enters the restart phase. There it performs a Type 3 pivot between x_9 and x_2. The resulting tableau is as follows.

$$10.7142 + 2.5714x_3 + 0.7142x_6 - 5.7142x_4 - 0.4285x_2 - 6.5714x_5 = C$$

$$-1.7142 + 5.4285x_3 - 0.7142x_6 - 5.2857x_4 + 5.4285x_2 - 1.4285x_5 = x_8$$

$$-2.5715 + 3.1428x_3 + 0.4285x_6 - 0.4285x_4 + 2.1428x_2 + 0.8571x_5 = x_7$$

$$2.1427 - 1.2857x_3 + 0.1428x_6 - 0.1428x_4 - 1.2857x_2 - 0.7142x_5 = x_1$$

$$12 - 6x_3 - 8x_2 - 6x_5 = x_9$$

The $\mathcal{L}(P^+)$ solution $(x_1, \cdots, x_5)$ = (0.7142, 0.0, 1.0, 1.0, 0.0) is primal infeasible, which means some basic variables violate their bound constraints. Here $x_8 = -1.5714$, thus violates the constraint that it is nonnegative. This provides $I_L(x^*) = 1.5714$. PC now checks whether complementing any single non-basic binary decision variables reduces $I_L(x^*)$. It finds that complementing x_4 reduces $I_L(x^*)$ to zero. The tableau does not change, but the solution $(x_1, \cdots, x_5)$ = (0.8571, 0.0, 1.0, 0.0, 0.0) is primal feasible. The solution is not P-feasible yet. At this point, PC again checks whether rounding or truncating of current solution gives a P-feasible solution. Rounding of the solution provides a P-feasible solution $(x_1, \cdots, x_5)$ = (1.0, 0.0, 1.0, 0.0, 0.0), and PC terminates successfully.

3.2 Pivot and Gomory Cut

As in PC, the initial goal of PGC is to find a P-feasible solution by bringing all the binary variables out of the simplex tableau basis. To do this, PGC uses pivoting rules similar to those of PC. The major difference between PGC and PC is that PGC uses Gomory cuts in this pivoting framework in the hope of finding a P-feasible solution as early as possible. PGC uses Gomory cuts for several purposes, namely to select pivots, to avoid cycling, and to replace the complementation based PC restart phase

We spent a long time searching for a good feasibility heuristic before arriving at PGC. Our search started with the implementation of PC. While experimenting with PC, we found that the restart phase often fails to bring execution back to the search phase. This led us to find a better way of moving to the $\mathcal{L}(P^+)$ feasible space from the dead end, namely by using a cutting plane. We selected Gomory cuts as our cutting plane because the PC pivoting framework uses a simplex tableau, and so generating Gomory cuts requires almost no extra cost. Later, we came up with the idea of also using Gomory cuts to guide the pivoting.

Here is how PGC works. It begins in the same manner as PC, namely by solving $\mathcal{L}(P^+)$ using

the bounded variable revised simplex method. If the optimal solution x^* of $\mathcal{L}(P^+)$ is not P-feasible and neither rounding nor truncating the fractional binary variables provides a P-feasible solution, PGC calls its feasibility subroutine, which, as in PC, consists of a search phase and restart phase.

Once execution reaches the search phase, PGC finds a Gomory cut separating x^* from the P-polytope. PGC then tries to decrease the extent to which x^* is P-infeasible by repeated tableau pivoting. During pivoting, whenever the current basic feasible solution x^* of $\mathcal{L}(P^+)$ satisfies the Gomory cut, PGC adds the cut to the formulation of $\mathcal{L}(P^+)$ to avoid cycling, and generates a new Gomory cut separating x^* from the P-polytope.

This generation and occasional addition of Gomory cuts to $\mathcal{L}(P^+)$ continues until the search phase either finds a P-feasible solution or reaches a dead end. In the latter case, PGC checks whether rounding or truncating the fractional binary variables yields a P-feasible solution. If not, execution transfers to the restart phase together with the most recently unsatisfied Gomory cut.

The goal of the restart phase is to find a basic feasible solution of $\mathcal{L}(P^+)$ that satisfies the Gomory cut and stays close to it. The intuition here is that, at the end of the search phase, the current basic feasible solution might be close to a P-feasible solution which in turn might be close to the Gomory cut. To find such a basic feasible solution, PGC replaces the objective function with the temporary objective function $max\ \alpha x$, where $\alpha x \geq \beta$ describes the cut. The bounded variable revised simplex algorithm continues until either the value of the temporary objective function is at least β or the nonbasic x_n chosen as the entering pivot variable has $max\ \alpha x$ unbounded or the algorithm fails to choose a nonbasic variable. In the last case, execution halts and declares that P is infeasible. In the other two cases, $\alpha x \geq \beta$ is added as a constraint to $\mathcal{L}(P^+)$, and so the surplus variable corresponding to this constraint becomes a basic variable x_b. In the case where the nonbasic x_n chosen as the entering pivot variable has $\max \alpha x$ unbounded, the simplex tableau becomes primal infeasible, so a pivot between x_n and x_b is performed to correct this. PGC then restores the original objective function and transfers execution to the search phase.

When P has a feasible solution, unlike the PC restart phase, which may not bring the execution back to the search phase, the PGC restart phase is guaranteed to bring the execution back to the search phase, although possibly at a cost of increasing primary P-infeasibility measure.

To generate Gomory cuts, considering x^* is the current basic feasible solution of $\mathcal{L}(P^+)$, PGC chooses the row of the simplex tableau corresponding to the most integer-infeasible binary variable x_j, where x_j is the basic binary variable for which the integer-infeasibility, measured by $min\{x_j^*, 1 - x_j^*\}$, is maximum. PGC then generates Gomory cuts from the chosen row following the construction shown in § 2.1.6.

Since we introduced a new criteria for choosing pivots, namely based on Gomory cuts, we implemented two variations of the PGC search phase in our experiments. One variation, PGC_0, uses the two pivot rules of the PC search phase. The other variation, PGC_1, uses two new pivot rules based on the distance between x^* and the Gomory cut.

In the rules given below, $\alpha x \geq \beta$ refers to the current cut, x^*_{cur} refers to the current basic feasible solution, and x^*_{next} refers to the basic feasible solution resulting from a pivot under consideration.

A P-feasible solution is always somewhere across the current Gomory cut. PGC_1 uses this fact to define its pivoting rules. The goal of the Type 1 pivot is to decrease the primary P-infeasibility measure, the number of basic binary variables, and cross the cut.

> PGC_1 *Type 1 Pivot*: The first pivot found that exchanges a basic binary decision variable with a non-basic continuous variable, maintains $\mathcal{L}(P^+)$-feasibility, and goes some distance towards the cut, namely $\beta - \alpha x^*_{next} < \beta - \alpha x^*_{cur}$.

Once there are no more such pivots, the goal becomes to keep the primary P-infeasibility measure unchanged and cross the cut.

> PGC_1 *Type 2 Pivot*: The pivot that exchanges a basic continuous variable with a non-basic continuous variable, or a basic binary decision variable with a nonbasic binary decision variable, maintains $\mathcal{L}(P^+)$-feasibility, and, among the first (at most) $\lfloor \log n \rfloor$ such pivots, the one which either crosses the cut by the smallest amount or, if no pivot crosses the cut, the one which is closest to the cut. Thus, if $\beta - \alpha x^*_{next} > 0.0$ for all the $\lfloor \log n \rfloor$ pivots, then select the pivot for which $\beta - \alpha x^*_{next}$ is minimum; otherwise, select the pivot for which $\beta - \alpha x^*_{next}$ is maximum over the choices for which $\beta - \alpha x^*_{next} \leq 0.0$.

In the selection of PGC_1 Type 2 pivot, we sample a relatively small number of pivots because experimental results showed the first eligible pivot, that keeps the primary P-infeasibility measure unchanged and goes some distance towards the cut, might not lead far towards the cut, whereas considering all (possibly n) eligible pivots might take too long.

Figure 3.1 shows a description of PGC algorithm. A more detailed version of the algorithm appears in Appendix A.

Algorithm PGC_0/PGC_1
INPUT: a 0-1 mixed integer problem P and a time limit T.
OUTPUT: a P-feasible solution x^* (null in case of failure).

```
1.  x* ← null;  elapsedTime ← 0.
2.  construct L(P+) from P.
3.  find optimal solution x* of L(P+) using bounded variable revised simplex algorithm.
4.  if( there is no solution to L(P+)) then
5.      return null with the the message that P is infeasible.
6.  if (x* or rounding of x* or truncation of x* is P-feasible) then
7.      return the P-feasible solution.
8.  repeat
    BEGIN SEARCH PHASE
9.      construct the Gomory Cut αx ≥ β
          from the row corresponding to the most integer-infeasible binary variable.
10.     atDeadEnd ← false.
11.     repeat
12.         while (a Type 1 Pivot of PGC0/PGC1 exists)
13.             perform the Type 1 Pivot;  x* ← resulting L(P+)-feasible solution.
14.             if (x* satisfies αx ≥ β) then
15.                 if (x* is P-feasible) then return x*.
16.                 else
17.                     add αx ≥ β in L(P+).
18.                     construct the new Gomory Cut αx ≥ β from the row
                          corresponding to the most integer-infeasible binary variable.
19.         if (a Type 2 Pivot of PGC0/PGC1 exists) then
20.             perform the Type 2 Pivot;  x* ← resulting L(P+)-feasible solution.
21.             if (x* satisfies αx ≥ β) then
22.                 if (x* is P-feasible) then return x*.
23.                 else
24.                     add αx ≥ β in L(P+).
25.                     construct the new Gomory Cut αx ≥ β from the row
                          corresponding to the most integer-infeasible binary variable.
26.         else atDeadEnd ← true.
27.     until atDeadEnd
    END SEARCH PHASE
28.     if (rounding or truncating binary variables gives P-feasible x*) then return x*.
    BEGIN RESTART PHASE
29.     replace the objective function "min cx" of L(P+) with "max αx".
30.     apply bounded variable revised simplex algorithm until
          the objective value is at least β.
31.     if (bounded variable revised simplex algorithm fails
          to find a solution whose objective value is at least β) then
32.         return null with the message that P is infeasible.
33.     re-establish the objective function to "min cx" and include αx ≥ β in L(P+).
    END RESTART PHASE
34. until (elapsedTime≥ T)
35. return x*.
```

Figure 3.1: A pseudo-code description of the algorithms PGC_0 and PGC_1.

Now we illustrate the execution of PGC_0 on the same example on which we illustrated the execution of PC. As in PC, PGC_0 first finds the following optimal simplex tableau.

$$5 + x_6 + x_7 + x_8 + x_9 = C$$
$$1.1978 - 1.0989x_2 + 0.2812x_6 - 0.5781x_7 + 0.0468x_8 - 0.2604x_9 = x_5$$
$$-0.1666 + 1.0833x_2 - 0.5x_6 + 0.75x_7 - 0.25x_8 + 0.1666x_9 = x_4$$
$$0.7083 - 0.3541x_2 + 0.375x_6 - 0.4375x_7 + 0.0625x_8 + 0.0416x_9 = x_1$$
$$0.4687 - 0.2343x_2 - 0.2812x_6 + 0.5781x_7 - 0.0468x_8 + 0.0937x_9 = x_3$$

The current solution $(x_1, x_2, x_3, x_4, x_5)$ = (0.3541, 1.0, 0.2343, 0.9166, 0.0989) is not P-feasible, and neither rounding nor truncating this solution yields a P-feasible solution. Therefore, PGC_0 enters the search phase of feasibility subroutine. There, since the measure of integer infeasibility for the variables x_1, x_2, x_3, x_4, and x_5 are 0.3541, 0, 0.2343, 0.0833, and 0.0989 respectively, PGC_0 generates a Gomory cut from the row corresponding to the most integer-infeasible binary variable, x_1. The generated Gomory cut has the following form.

$$0.1942x_2 + 0.2056x_6 + 0.4375x_7 + 0.0342x_8 + 0.0228x_9 \geq 0.3541.$$

PGC_0 performs the same pivots that PC did until this Gomory cut is crossed. As illustrated in the previous section, performing a Type 1 pivot between x_7 and x_4, a Type 1 pivot between x_9 and x_5, a Type 2 pivot between x_8 and x_9, and a Type 2 pivot between x_2 and x_3 bring PGC_0 to the following simplex tableau.

$$10.1785 + 2.8928x_3 + 0.7142x_6 - 5.7142x_4 + 0.0535x_9 - 6.25x_5 = C$$
$$5.0718 + 1.3571x_3 - 0.7142x_6 - 5.2857x_4 - 0.6785x_9 - 5.5x_5 = x_8$$
$$0.107 + 1.5357x_3 + 0.4285x_6 - 0.4285x_4 - 0.2678x_9 - 0.75x_5 = x_7$$
$$0.5356 - 0.3214x_3 + 0.1428x_6 - 0.1428x_4 + 0.1607x_9 + 0.25x_5 = x_1$$
$$1.25 - 0.75x_3 - 0.125x_9 - 0.75x_5 = x_2$$

At this point, although the current solution $(x_1, \cdots, x_5)$ = (0.0714, 0.5, 1.0, 1.0, 0.0) is not yet P-feasible, it has crossed the Gomory cut. Therefore, PGC_0 adds the Gomory cut to the simplex tableau. This yields the following tableau, where x_{10} is the new surplus variable corresponding to the Gomory cut.

$$10.1785 + 2.8928x_3 + 0.7142x_6 - 5.7142x_4 + 0.0535x_9 - 6.25x_5 = C$$

$$5.0718 + 1.3571x_3 - 0.7142x_6 - 5.2857x_4 - 0.6785x_9 - 5.5x_5 = x_8$$

$$0.107 + 1.5357x_3 + 0.4285x_6 - 0.4285x_4 - 0.2678x_9 - 0.75x_5 = x_7$$

$$0.5356 - 0.3214x_3 + 0.1428x_6 - 0.1428x_4 + 0.1607x_9 + 0.25x_5 = x_1$$

$$1.25 - 0.75x_3 - 0.125x_9 - 0.75x_5 = x_2$$

$$-0.182 + 0.864x_3 + 0.3686x_6 - 0.3686x_4 - 0.0933x_9 - 0.3709x_5 = x_{10}$$

Next, since the measure of integer infeasibility for the variables x_1, x_2, x_3, x_4, and x_5 are 0.0714, 0.5, 0, 0, and 0 respectively, PGC_0 constructs the following new Gomory cut from the row corresponding to the most integer-infeasible binary variable, x_2.

$$0.25x_3 + 0.125x_9 + 0.25x_5 \geq 0.5.$$

As there is no Type 1 or Type 2 pivot from this tableau, PGC_0 checks whether rounding or truncating the current solution yields a P-feasible solution. Since neither process yields a P-feasible solution, PGC_0 enters the restart phase. There, it replaces the current objective function with *max* $0.25x_3 + 0.125x_9 + 0.25x_5$. It applies the bounded variable revised simplex algorithm until the cut is crossed, namely until the objective value is at least 0.5. PGC_0, then restores the previous objective function and adds the crossed Gomory cut to the tableau. This yields the following simplex tableau, where x_{11} is the new surplus variable corresponding to the added Gomory cut.

$$1.2143 + 6.7857x_1 - 1.5x_2 + 4x_4 + 1.5x_8 + 2.2142x_{10} = C$$

$$-0.4881 + 0.9881x_1 + 1.25x_2 - 0.5x_4 - 0.25x_8 + 1.8452x_{10} = x_6$$

$$0.8095 - 0.8095x_1 + 1.4761x_{10} = x_7$$

$$-4.1071 + 6.6071x_1 - 2.75x_2 + 4.5x_4 + 0.75x_8 - 1.1071x_{10} = x_9$$

$$-0.52 + 0.6138x_1 - 0.1406x_2 + 0.0937x_4 + 0.0156x_8 - 0.2075x_{10} = x_{11}$$

$$0.6889 - 0.1264x_1 - 0.8437x_2 + 0.5625x_4 + 0.0937x_8 + 0.2306x_{10} = x_3$$

$$1.6622 - 0.9747x_1 - 0.0312x_2 - 1.3125x_4 - 0.2187x_8 - 0.0461x_{10} = x_5$$

The search phase then resumes. The solution $(x_1, \cdots, x_5) = (1.0, 0.0, 0.5625, 0.0, 0.6875)$ is not yet P-feasible, and since the measure of integer infeasibility for the variables x_1, x_2, x_3, x_4, and x_5 are 0, 0, 0.4375, 0, and 0.3125 respectively, PGC_0 generates the following Gomory cut from the row corresponding to the most integer-infeasible binary variable, x_3.

$$0.1626x_1 + 0.2008x_2 + 0.4375x_4 + 0.1205x_8 + 0.2965x_{10} \geq 0.5625$$

There is no available Type 1 pivot, so PGC_0 performs a Type 2 pivot between x_8 and x_6. This yields the following simplex tableau.

$$\begin{aligned}
-1.7142 + 12.7142x_1 + 6x_2 + x_4 - 6x_6 + 13.2857x_{10} &= C \\
-1.9524 + 3.9523x_1 + 5x_2 - 2x_4 - 4x_6 + 7.3809x_{10} &= x_8 \\
0.8095 - 0.8095x_1 + 1.4761x_{10} &= x_7 \\
-5.5714 + 9.5714x_1 + x_2 + 3x_4 - 3x_6 + 4.4285x_{10} &= x_9 \\
-0.5505 + 0.6755x_1 - 0.0625x_2 + 0.0625x_4 - 0.0625x_6 - 0.0922x_{10} &= x_{11} \\
0.506 + 0.2440x_1 - 0.375x_2 + 0.375x_4 - 0.375x_6 + 0.9226x_{10} &= x_3 \\
2.0892 - 1.8392x_1 - 1.125x_2 - 0.875x_4 + 0.875x_6 - 1.6607x_{10} &= x_5
\end{aligned}$$

The solution $(x_1, \cdots, x_5) = (1.0, 0.0, 0.75, 0.0, 0.25)$ is not yet P-feasible, and has not yet crossed the cut. PGC_0 finds a Type 1 pivot between x_{10} and x_5. Performing the pivot yields the following simplex tableau.

$$\begin{aligned}
15 - 2x_1 - 3x_2 - 6x_4 + x_6 - 8x_5 &= C \\
7.3333 - 4.2222x_1 - 5.8888x_4 - 0.1111x_6 - 4.4444x_5 &= x_8 \\
2.6666 - 2.4444x_1 - x_2 - 0.7777x_4 + 0.7777x_6 - 0.8888x_5 &= x_7 \\
4.6666x_1 - 2x_2 + 0.6666x_4 - 0.6666x_6 - 2.6666x_5 &= x_9 \\
-0.6666 + 0.7777x_1 + 0.1111x_4 - 0.1111x_6 + 0.0555x_5 &= x_{11} \\
1.6666 - 0.7777x_1 - x_2 - 0.1111x_4 + 0.1111x_6 - 0.5555x_5 &= x_3 \\
1.2581 - 1.1075x_1 - 0.6774x_2 - 0.5268x_4 + 0.5268x_6 - 0.6021x_5 &= x_{10}
\end{aligned}$$

The solution $(x_1, \cdots, x_5) = (1.0, 0.0, 0.8888, 0.0, 0.0)$ is not yet P-feasible, and has not yet crossed the cut. PGC_0 again finds a Type 1 pivot between x_6 and x_3. Performing the pivot yields the following simplex tableau.

$$5x_1 + 6x_2 - 5x_4 + 9x_3 - 3x_5 = C$$

$$9 - 5x_1 - x_2 - 6x_4 - x_3 - 5x_5 = x_8$$

$$-9 + 3x_1 + 6x_2 + 7x_3 + 3x_5 = x_7$$

$$10 - 8x_2 - 6x_3 - 6x_5 = x_9$$

$$1 - x_2 - x_3 - 0.5x_5 = x_{11}$$

$$-15 + 7x_1 + 9x_2 + x_4 + 9x_3 + 5x_5 = x_6$$

$$-6.6451 + 2.5806x_1 + 4.0645x_2 + 4.7419x_3 + 2.0322x_5 = x_{10}$$

Finally, the solution $(x_1, \cdots, x_5) = (1.0, 0.0, 1.0, 0.0, 0.0)$ is P-feasible. Therefore, PGC_0 terminates successfully.

Now we illustrate the execution of PGC_1, where the pivoting rules are different from PGC_0, on the same example. PGC_1 starts in the same way as PGC_0 by finding the optimal simplex tableau and entering the search phase where it finds a Gomory cut from the row corresponding to the most integer-infeasible binary variable, x_1. Then it looks for a PGC_1 Type 1 pivot, and it finds one between x_6 and x_3. Performing this pivot yields the following tableau.

$$6.6666 - 0.8333x_2 - 3.5555x_3 + 3.0555x_7 + 0.8333x_8 + 1.3333x_9 = C$$

$$1.6666 - 1.3333x_2 - x_3 - 0.1666x_9 = x_5$$

$$-1 + 1.5x_2 + 1.7777x_3 - 0.2777x_7 - 0.1666x_8 = x_4$$

$$1.3333 - 0.6666x_2 - 1.3333x_3 + 0.3333x_7 + 0.1666x_9 = x_1$$

$$1.6666 - 0.8333x_2 - 3.5555x_3 + 2.0555x_7 - 0.1666x_8 + 0.3333x_9 = x_6$$

The solution $(x_1, \cdots, x_5) = (0.6666, 1.0, 0.0, 0.5, 0.3333)$ is not yet P-feasible, and has not crossed the cut. PGC_1 finds a Type 1 pivot between x_9 and x_5. Performing the pivot yields the following simplex tableau.

$$20 - 11.5x_2 - 11.5555x_3 + 3.0555x_7 + 0.8333x_8 - 8x_5 = C$$

$$10 - 8x_2 - 6x_3 - 6x_5 = x_9$$

$$-1 + 1.5x_2 + 1.7777x_3 - 0.2777x_7 - 0.1666x_8 = x_4$$

$$3 - 2x_2 - 2.3333x_3 + 0.3333x_7 - x_5 = x_1$$

$$5 - 3.5x_2 - 5.5555x_3 + 2.0555x_7 - 0.1666x_8 - 2x_5 = x_6$$

The solution $(x_1, \cdots, x_5) = (1.0, 1.0, 0.0, 0.5, 0.0)$ is not yet P-feasible but has crossed the cut. Therefore, PGC_1 adds the crossed cut to the simplex tableau. This yields the following tableau, where x_{10} is the new surplus variable corresponding to the Gomory cut.

$$20 - 11.5x_2 - 11.5555x_3 + 3.0555x_7 + 0.8333x_8 - 8x_5 = C$$

$$10 - 8x_2 - 6x_3 - 6x_5 = x_9$$

$$-1 + 1.5x_2 + 1.7777x_3 - 0.2777x_7 - 0.1666x_8 = x_4$$

$$3 - 2x_2 - 2.3333x_3 + 0.3333x_7 - x_5 = x_1$$

$$5 - 3.5x_2 - 5.5555x_3 + 2.0555x_7 - 0.1666x_8 - 2x_5 = x_6$$

$$1.0967 - 1.0967x_2 - 1.2795x_3 + 0.8602x_7 - 0.5483x_5 = x_{10}$$

Since the measure of integer infeasibility for the variables x_1, x_2, x_3, x_4, and x_5 are 0, 0, 0, 0.5, and 0 respectively, PGC_1 constructs the following Gomory cut from the row corresponding to the most integer-infeasible binary variable, x_4.

$$0.5x_2 + 0.2222x_3 + 0.2777x_7 + 0.1666x_8 \geq 0.5.$$

PGC_1 again looks for a Type 1 pivot, and it finds one between x_8 and x_4. Performing this pivot yields the following tableau.

$$15 - 4x_2 - 2.6666x_3 + 1.6666x_7 - 5x_4 - 8x_5 = C$$

$$10 - 8x_2 - 6x_3 - 6x_5 = x_9$$

$$-6 + 9x_2 + 10.6666x_3 - 1.6666x_7 - 6x_4 = x_8$$

$$3 - 2x_2 - 2.3333x_3 + 0.3333x_7 - x_5 = x_1$$

$$6 - 5x_2 - 7.3333x_3 + 2.3333x_7 + x_4 - 2x_5 = x_6$$

$$1.0967 - 1.0967x_2 - 1.2795x_3 + 0.8602x_7 - 0.5483x_5 = x_{10}$$

Finally, the solution $(x_1, \cdots, x_5) = (1.0, 1.0, 0.0, 0.0, 0.0)$ is P-feasible. Therefore, PGC_1 terminates successfully.

Although we have described PGC as a 0-1 mixed integer programming heuristic, it is easily modified to be a heuristic for general mixed integer programs. It suffices to replace in our description of PGC the term 'binary' with the term 'integer'. To illustrate, we show the execution of PGC_1 on the following simple general mixed integer program P.

$$\min \{ x_1 + x_2 \mid 6x_1 + 4x_2 \geq 9,\ 3x_1 - 4x_2 \leq 3,\ 3x_1 + 4x_2 \leq 18,\ x_1, x_2 \in \mathbb{Z}_+^2 \}.$$

First, PGC_1 adds slack/surplus variables to create the following P^+.

$$\min \{ C = x_1 + x_2 \mid 6x_1 + 4x_2 - x_3 = 9,\ 3x_1 - 4x_2 + x_4 = 3,$$

$$3x_1 + 4x_2 + x_5 = 18,\ x_1, x_2 \in \mathbb{Z}^2_+,\ x_3, x_4, x_5 \in \mathbb{R}^3_+ \}.$$

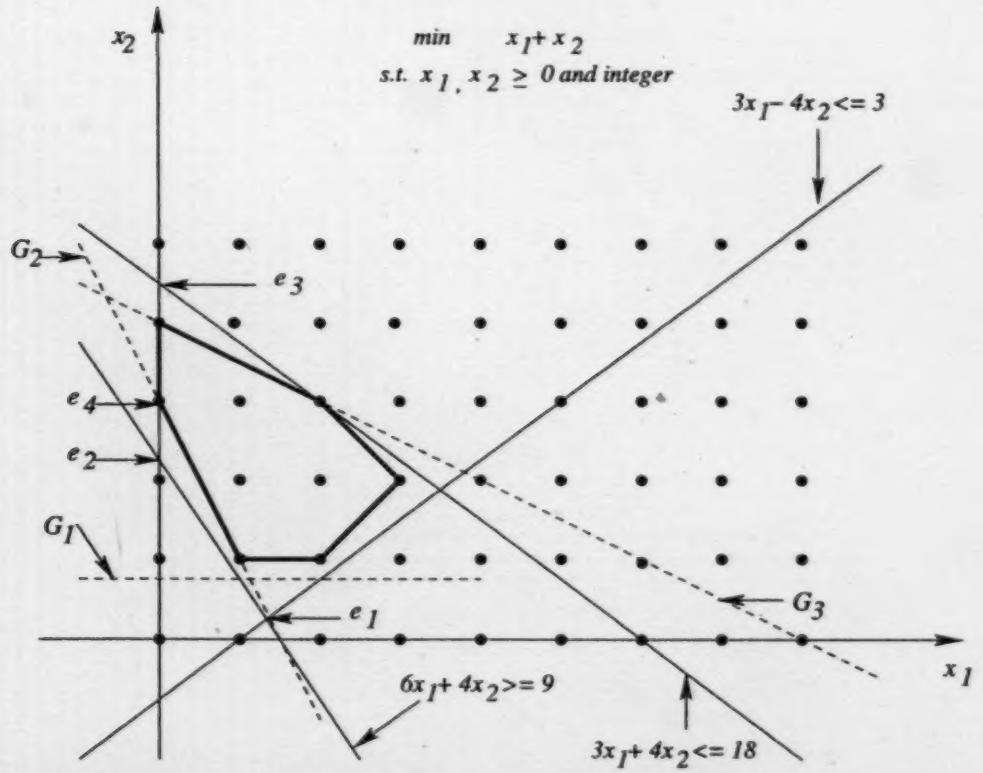

Figure 3.2: Illustration of PGC_1 on a small example. Thin lines show the original constraints. Bold lines show the integer polytope. Dashed lines show the Gomory cuts PGC uses.

Next, PGC_1 finds the optimal solution $e_1 = (\frac{4}{3}, \frac{1}{4})$ of the $\mathcal{L}(P^+)$ shown in Figure 3.2. The simplex tableau corresponding to e_1 has the following form, where x_3, x_4 are the nonbasic variables, and x_1, x_2, and x_5 are the basic variables.

$$\tfrac{19}{12} + \tfrac{7}{36}x_3 + \tfrac{1}{18}x_4 = C$$

$$\tfrac{4}{3} + \tfrac{1}{9}x_3 - \tfrac{1}{9}x_4 = x_1$$

$$\tfrac{1}{4} + \tfrac{1}{12}x_3 + \tfrac{1}{6}x_4 = x_2$$

$$13 - \tfrac{2}{3}x_3 - \tfrac{1}{3}x_4 = x_5$$

Since neither e_1 nor a rounding or truncating of e_1 is P-feasible, PGC_1 selects the tableau row, corresponding to the most integer-infeasible integer-constrained variable at e_1, to generate a Gomory cut. Since the measure of integer infeasibility for the variables x_1 and x_2 are $\frac{1}{3}$ and $\frac{1}{4}$ respectively,

PGC_1 generates the Gomory cut $\frac{1}{18}x_3 + \frac{1}{9}x_4 \geq \frac{1}{3}$ from the tableau row corresponding to x_1. This Gomory cut is equivalent to $x_2 \geq \frac{3}{4}$ in the space of variables x_1 and x_2. Figure 3.2 shows this cut as G_1. PGC_1 then looks for a Type 1 pivot and finds one between x_1 and x_4. Performing this pivot brings PGC_1 to $e_2 = (0, \frac{9}{4})$. Since PGC_1 has crossed the Gomory cut G_1 and e_2 is not P-feasible, PGC_1 adds the crossed cut G_1 as a new row, introducing a new surplus variable x_6, in the simplex tableau. The simplex tableau now has the following form, where x_1, x_3 are the nonbasic variables, and x_2, x_4, x_5, and x_6 are the basic variables.

$$\frac{9}{4} - \frac{1}{2}x_1 + \frac{1}{4}x_3 = C$$

$$\frac{9}{4} - \frac{3}{2}x_1 + \frac{1}{4}x_3 = x_2$$

$$12 - 9x_1 + x_3 = x_4$$

$$9 + 3x_1 - x_3 = x_5$$

$$1 - x_1 + \frac{1}{6}x_3 = x_6$$

Since e_2 is not P-feasible and PGC_1 has no Gomory cut in hand that is violated by e_2, PGC_1 generates a new Gomory cut from the tableau row corresponding to the most integer-infeasible integer-constrained variable. Since the measure of integer infeasibility for the variables x_1 and x_2 are 0 and $\frac{1}{4}$ respectively, PGC_1 generates the Gomory cut $\frac{1}{6}x_1 + \frac{1}{12}x_3 \geq \frac{1}{4}$ from the tableau row corresponding to x_2. This Gomory cut is equivalent to $2x_1 + x_2 \geq 3$ in the space of variables x_1 and x_2. Figure 3.2 shows this cut as G_2. PGC_1 then looks for a Type 1 pivot, and as there is no such pivot available, it looks for a Type 2 pivot. There exists only one Type 2 pivot between x_5 and x_3. Performing this pivot brings PGC_1 to $e_3 = (0, \frac{9}{2})$. Since PGC_1 has crossed the Gomory cut G_2 and e_3 is not P-feasible, PGC_1 adds the cut G_2 as a new row, introducing a new surplus variable x_7, in the simplex tableau. The simplex tableau now has the following form, where x_1, x_5 are the nonbasic variables, and x_2, x_3, x_4, x_6, and x_7 are the basic variables.

$$\frac{9}{2} + \frac{1}{4}x_1 - \frac{1}{4}x_5 = C$$

$$\frac{9}{2} - \frac{3}{4}x_1 - \frac{1}{4}x_5 = x_2$$

$$9 + 3x_1 - x_5 = x_3$$

$$21 - 6x_1 - x_5 = x_4$$

$$\frac{5}{2} - \frac{1}{2}x_1 - \frac{1}{6}x_5 = x_6$$

$$\frac{1}{2} + \frac{5}{12}x_1 - \frac{1}{12}x_5 = x_7$$

Since e_3 is not P-feasible and PGC_1 has no Gomory cut in hand that is violated by e_3, PGC_1 generates new Gomory cut from the tableau row corresponding to the most integer-infeasible integer-constrained variable. Since the measure of integer infeasibility for the variables x_1 and x_2 are 0 and $\frac{1}{2}$ respectively, PGC_1 generates the Gomory cut $\frac{1}{4}x_1 + \frac{1}{4}x_5 \geq \frac{1}{2}$ from the tableau row corresponding to x_2. This Gomory cut is equivalent to $x_1 + 2x_2 \geq 8$ in the space of variables x_1 and x_2. Figure 3.2 shows this cut as G_3. PGC_1 then looks for a Type 1 pivot, and as there is no such pivot available, it looks for a Type 2 pivot. There exists only one Type 2 pivot between x_7 and x_5. Performing this pivot brings PGC_1 to $e_4 = (0, 3)$. Since PGC_1 has crossed the Gomory cut G_3 and e_4 is P-feasible, PGC_1 terminates successfully returning e_4 as a P-feasible solution.

In the previous example, PGC_1 does not reach a dead end. But in instances with a higher number of integer-constrained variables, PGC_1 is more likely to reach a dead end. When it does, it overcomes that situation by crossing the cut as described in the restart phase.

In this section, we have seen the details of PGC. In the next sections, we see how PGC performs.

3.3 Heuristic Performance Evaluation

Recent heuristics evaluate their performance on a set of benchmark instances, namely instances from the libraries MIPLIB and DEISLIB. These libraries contain one or two instances of many different kinds of optimization problems. For testing purposes, it is of interest to be able to generate more instances of these kinds of problems. In this thesis, we compared our heuristics on the available benchmark instances as well as on two classes of pseudo-randomly generated hard 0-1 integer program instances. One class of instances is from our introduced constrained market-sharing problem described in Chapter 6. The other is from Cornuéjols-Dawande feasibility and optimality model of the market-sharing problem described in § 2.3.2.

Recent feasibility heuristics such as FP, the new implementation of PS, evaluated their performance only against commercial solvers such as Cplex and Xpress, rather than against each other. Similarly, recent improvement heuristics (with the exception of RINS, which compared itself against both the commercial solver Cplex as well as the heuristic LB) evaluated their performance only against the commercial solvers.

In this thesis, we have evaluated the performance of our new heuristics both against some commercial solver and some comparable recent non-commercial heuristics as well as any previous heuristic on which our new heuristic is based.

3.4 PGC Performance Evaluation

To evaluate the performance of PGC, we compared it against three solvers: FP, the recent feasibility heuristic, PC, the heuristic on which PGC is based, and Cplex Version 9.13, a commercial mixed integer programming solver. We also compared PGC against PBS Version 4.0, a state of the art pseudo-boolean solver, on the 0-1 integer program instances.

From the benchmark suite shown in Table B.1- B.3 of Appendix B, we chose all 77 0-1 mixed integer program instances with the exception of instance 'stp3d'. We omitted this instance because the GLPK linear programming solver cannot handle this instance, and a feasible solution for this instance is yet unknown. We also chose some pseudo-randomly generated instances of two different kinds of problems. One class is from the constrained market-sharing problem defined in Chapter 6. The other is from the Cornuéjols-Dawande feasibility model of market-sharing problem described in § 2.3.2.

We ran all experiments on a 2403 MHz AMD Athlon processor with 128 MByte of memory under Redhat Linux 9.0. We implemented PC, both versions of PGC, and FP in the C programming language on top of version 4.0 of the open source GLPK. We alloted one CPU-hour to each solver for finding a feasible solution of a particular instance.

In the following sections, we see the comparison of PGC against the considered solvers.

3.4.1 PGC_0 versus PGC_1 versus PC

Since we consider PC as the predecessor of PGC, we first compare both versions of PGC against our implementation of PC.

Table 3.1 and 3.2 summarize the performance of PGC_0 and PGC_1 against PC. For detailed experimental results of PC, PGC_0, and PGC_1, see Table B.4 in Appendix B.

From Table 3.1 and 3.2, we see that both PGC_0 and PGC_1 are much more successful than PC in finding some feasible solution. Among the instances in which PC succeeds, PGC_1 is faster to find a solution more often than PC, whereas PC finds a better solution more often. Since our prime objective is to find a feasible solution as soon as possible, PGC_1 outperforms PC on the chosen benchmark instances with respect to both success rate and time.

Table 3.1: PC versus PGC_0 on 77 benchmark instances. Entries indicate number of instances.

	PC	PGC_0
successful	29	62
among the 29 instances in which PC succeeds		
takes lesser amount of time in	9	9
takes equal amount of time in	11	
finds better solution in	15	6
finds same solution in	8	

Table 3.2: PC versus PGC_1 on 77 benchmark instances. Entries indicate number of instances.

	PC	PGC_1
successful	29	62
among the 29 instances in which PC succeeds		
takes lesser amount of time in	5	17
takes equal amount of time in	7	
finds better solution in	24	4
finds same solution in	1	

This experiment also shows the advantage of PGC restart phase in the way that PGC restart phase always brings execution back to the search phase, whereas PC restart phase failed to bring execution back to the search phase in 35 of the 77 instances. PGC restart phase is guaranteed to bring the execution back to the search phase since the simplex method applied at the search phase

is guaranteed to cross the cut, whereas the complementation of one or two nonbasic variables in PC may fail to bring the execution back to the search phase.

PGC_0 and PGC_1 succeed on the same 62 instances. Table 3.3 summarizes the performance of only PGC_0 and PGC_1. This table supports our belief that the Gomory cut guided pivot rules of PGC_1 are more effective in finding feasible solutions quickly than the PC-pivot rules of PGC_0, which are focused on good objective value.

Table 3.3: PGC_0 versus PGC_1 on the 62 instances in which both succeed. Entries indicate number of instances.

	PGC_0	PGC_1
takes lesser amount of time in	11	33
takes equal amount of time in	18	
finds better solution in	27	16
finds same solution in	19	

Since PGC_1 outperforms PGC_0 with respect to finding a feasible solution quickly, we pick PGC_1 as the version of PGC to compare against other solvers.

3.4.2 PGC_1 versus Feasibility Pump

Table 3.4 summarizes the performance of FP and PGC_1. For detailed results on FP and PGC_1, see Table B.5 in Appendix B. Results shown in Table 3.4 suggest that PGC_1 is a competitive alternative of FP in finding a feasible solution as quickly as possible.

Table 3.4: FP versus PGC_1 on 77 benchmark instances. Entries indicate number of instances.

	FP	PGC_1
successful	63	62
among the 66 instances in which at least one succeeds		
takes lesser amount of time in	26	29
takes equal amount of time in	11	
among the 59 instances in which both succeed		
finds better solution in	28	17
finds same solution in	14	

3.4.3 PGC_1 versus ILOG Cplex 9.13

In order to show how PGC_1 compares against state-of-the-art mixed integer program solvers, we used ILOG Cplex 9.13 mixed integer program solver in its default setup, called Cplex-D, as well as the setup with the emphasis for finding feasible solutions as early as possible, called Cplex-F.

Before showing the comparison against Cplex 9.13 mixed integer program solver, it is to be noted that the Cplex mixed integer program solver uses the Cplex linear programming solver, which is considerably faster than the GLPK linear programming solver on which we implemented PGC and FP. Table B.6 in Appendix B shows the comparison of the Cplex linear programming solver against the GLPK linear programming solver. We used GLPK linear programming solver since we had the access to the entire code of GLPK; this allowed us to implement PGC using the data structures of GLPK. For example, we had access to the GLPK allocated data structures to access the simplex tableau; we had direct access to the parameters of basic solutions. In contrast, we did not have access to the Cplex code, and we would need to re-allocate memory for the data structures and would need to make function calls for the necessary parameters of basic solutions. Since implementing PGC requires accessing the simplex tableau and other parameters of the given integer program frequently, we did not choose Cplex to implement PGC using the functions available to the Cplex users. The

overhead associated with the function calls and the re-allocation of the data structures would not reflect the true performance of PGC.

Table 3.5: Cplex-D versus PGC_1 on 77 benchmark instances. Entries indicate number of instances.

	Cplex-D	PGC_1
successful	71	62
among the 71 instances in which either one succeeds		
takes lesser amount of time in	45	15
takes equal amount of time in	11	
among the 62 instances in which both succeed		
finds better solution in	49	8
finds same solution in	5	

Table 3.6: Cplex-F versus PGC_1 on 77 benchmark instances. Entries indicate number of instances.

	Cplex-F	PGC_1
successful	73	62
among the 73 instances in which either one succeeds		
takes lesser amount of time in	47	17
takes equal amount of time in	9	
among the 62 instances in which both succeed		
finds better solution in	46	12
finds same solution in	4	

Table 3.5 and Table 3.6 summarizes the performance of PGC_1 against Cplex-D and Cplex-F respectively. For detailed results, see Table B.7 in Appendix B.

In spite of using GLPK linear programming solver, PGC_1 takes equal or less amount of time to find a feasible solution comparing to the Cplex in about one third instances of the benchmark suite.

3.4.4 PGC versus a Pseudo-Boolean Solver

In our chosen 77 0-1 mixed integer program benchmark instances, there are 10 instances that are 0-1 integer programs. We compare PGC_1 against pseudo-boolean solver PBS4 on these 10 instances. Table 3.7 summarizes the performance of PGC_1 against PBS4. For detailed result, see Table B.8 in Appendix B.

Table 3.7: PBS4 versus PGC_1 on 10 0-1 integer programming benchmark instances. Entries indicate number of instances.

	PBS4	PGC_1
successful	8	8
among the 10 instances in which either one succeeds		
takes lesser amount of time in	5	5
among the 6 instances in which both succeed		
finds better solution in	0	6

The results shown in Table 3.7 suggest that, in finding feasible solutions quickly, PGC_1 is competitive to PBS4 on this set of instances.

3.4.5 Performance on Randomly Generated Instances

We now show the performance of different solvers on a set of pseudo-randomly generated Cornuéjols-Dawande feasibility-hard instances described in § 2.3.2.

For instances with n variables and m constraints, Cornuéjols and Dawande showed that picking the relation $n = 10(m-1)$ yielded the hardest instances for their optimality model of market-sharing problem. Later, Aardal et al. showed that the Cornuéjols-Dawande feasibility-hard instances, generated with $n = 10(m-1)$ are with high probability infeasible. However, it follows easily from their analysis that choosing different values for n and m yields Cornuéjols-Dawande feasibility-hard instances that are with high probability feasible.

Table 3.8 shows the probability measures, namely the probability of a generated instance being infeasible and the expected number of solutions of a generated instance, of Cornuéjols-Dawande feasibility-hard instances for some n and m.

Table 3.8: Probability measures for the Cornuéjols-Dawande feasibility-hard instances generated with different n and m. The values are obtained using the analysis of Aardal et al. [1].

Problem size		Probability of	Expected number
n	m	being infeasible	of solutions
10	2	0.971	0.029
15	2	0.535	0.624
20	2	2.89e-07	15.056
20	3	0.925	0.077
25	3	0.169	1.778
30	3	1.39e-19	43.414
30	4	0.826	0.191
35	4	0.011	4.509
40	4	8.407e-49	110.69

Since our objective is to evaluate the performance of feasibility heuristics, the generated instances should have at least one feasible solution and finding any such solution should be hard. Therefore, based on the values shown in Table 3.8, we chose $n = 10m$ to generate Cornuéjols-Dawande feasibility-hard instances. Notice that for this choice of n and m the probability of a generated instance being infeasible is close to zero. Also, to indicate that these instances are feasibility hard, the expected number of solutions for the generated instance is found to be very low compared to 2^n, the total size of the enumeration space.

Using $n = 10m$, we pseudo-randomly generated five Cornuéjols-Dawande feasibility-hard instances with 10 to 50 variables each. The experimental results presented in Table B.9 of Appendix B show that both Cplex-D and Cplex-F perform better than other considered solvers on this set of instances. PBS4 performs well for smaller instances but becomes worse as the instance size grows. Between FP and PGC_1, there is no clear winner on this set of instances.

While PGC_1 is much worse than Cplex in the Cornuéjols-Dawande feasibility-hard instances, we now show a set of instances where PGC_1 is much stronger than Cplex. We do not know the reason for this difference in PGC's performances. In Chapter 6, we suggest some possible explanations.

We show that PGC_1 is much stronger than Cplex in a set of pseudo-randomly generated constrained market-sharing instances presented in Chapter 6. We generated instances from three groups. A parameter k differentiates these groups, where k is introduced to relate the number of variables n and the number of constraints m of instances. The relation between n and m is defined by $m = \lfloor \frac{n}{k} \rfloor$.

In the first group we set $k = 2.0$ and generated five pseudo-random instances of the problem with 50 to 400 variables each. Table B.10 of Appendix B shows the performance of different solvers on this set of pseudo-randomly generated instances.

In the second group we set $k = 1.5$ and generated five pseudo-random instances of the problem with 50 to 200 variables each. Table B.11 of Appendix B shows the performance of different solvers on this set of pseudo-randomly generated instances.

In the third group we set $k = 1.3$ and generated five pseudo-random instances of the problem

with 50 to 150 variables each. Table B.12 of Appendix B shows the performance of different solvers on this set of pseudo-randomly generated instances.

Results shown in Table B.10- B.12 of Appendix B suggest that PGC_1 outperforms all the considered solvers on this set of instances within the time limit of one CPU-hour. In Chapter 6, where we introduce this set of instances as a new class of hard 0-1 integer program instances, we present results of a large-scale experiment. In the large-scale experiment, we generated 100 instances of each size instead of generating only 5 instances. However, the performance of different solvers remained similar in the instances of the large-scale experiment.

3.4.6 Weakness of PGC

Our experiments revealed some weaknesses in the PGC search phase. It often reaches a dead end without having improved feasibility, namely without finding many pivots of Type 1 or Type 2. This happened in instances such as 10teams, ds, net12, protfold, t17171, swath. Also, execution sometimes terminated without reaching a dead end, namely finding too many pivots of Type 1 and Type 2. This happened in instances such as dano3mip, momentum1, nsr8k, rail4284c, rail4872c, and siena1.

Possible remedies of these flaws include the termination of the search phase after a certain number of pivots and the application of a neighbourhood search around the extreme point at that stage. We incorporated these ideas in another new 'find and improve' type heuristic NPGC, which we describe in Chapter 5.

3.5 Complexity of PGC

To show that PGC successfully terminates in a finite number of steps, we need to show that PGC performs a finite number of different pivots and uses a finite number of Gomory cuts. The number of Type 1 pivots in a search phase is bounded by the number of binary decision variables. However, the number of Type 2 pivots is not bounded by a finite number as long as the number of Gomory cuts added in PGC is not shown to be finite. Though Gomory, in his cutting plane algorithm [40], showed that there is a way of adding Gomory cuts that lead to an optimal solution in a finite number

of steps if the objective function is integer valued, we are unable to show that the way Gomory cuts are added in PGC ensures finding a feasible solution in a finite number of steps.

Chapter 4

Distance Induced Neighbourhood Search

In this chapter we present DISTANCE INDUCED NEIGHBOURHOOD SEARCH (DINS), a new mixed integer programming improvement heuristic. DINS is based on neighbourhood search. It defines a promising search neighbourhood around a known feasible solution at different nodes of a mixed integer program search tree generated by either a branch-and-bound or a branch-and-cut solver, and searches that neighbourhood with either a branch-and-bound or a branch-and-cut solver.

Recall that the LOCAL BRANCHING [35] and RELAXATION INDUCED NEIGHBOURHOOD SEARCH [29] heuristics described in § 2.2.11 and § 2.2.12 define such search neighbourhoods in two different ways, namely by respectively soft fixing and hard fixing integer-constrained variables. DINS defines its search neighbourhood by using a metric that measures at a node of a mixed integer program search tree the distance between the current mixed integer program solution and the node's associated relaxation solution.

As in the implementation of LB and RINS by Danna et al. we use the commercial solver Cplex both to generate the mixed integer program search tree and to search the neighbourhoods.

In the next sections, we see the details of DINS.

4.1 Distance Induced Neighbourhood Search

In § 2.1 and § 3.1 we gave the formulation of an integer program and a 0-1 mixed integer program respectively. We now give the analogous formulation of a general mixed integer program. We assume that the input program P is a mixed integer program of the form shown below, where c, x, b, A have dimensions $n, n, m, m \times n$ respectively, $N = \{1, \ldots, n\}$ is the set of variable indices of P which is partitioned into $(\mathcal{B}, \mathcal{G}, \mathcal{C})$ with $\mathcal{B}$, $\mathcal{G}$, and $\mathcal{C}$ denoting the indices of binary, general integer, and continuous variables respectively, and, for each index j for which x_j is a non-binary variable, l_j and u_j denote the respective lower and upper bounds for x_j. An integer-constrained variable is any variable in $\mathcal{B} \cup \mathcal{G}$.

$$P\text{: min } \{\, cx \mid Ax \geq b,\ x_i \in \{0,1\}\ \forall i \in \mathcal{B},$$

$$x_j \in \mathbb{Z} \text{ and } l_j \leq x_j \leq u_j\ \forall j \in \mathcal{G}, \qquad l_j \leq x_j \leq u_j\ \ \forall j \in \mathcal{C}\}.$$

In contrast to RINS, which performs only hard fixing of arbitrary variables, and LB, which performs only soft fixing of integer-constrained variables, DINS incorporates some hard fixing, some soft fixing, and some rebounding of integer-constrained variables. Furthermore, in DINS all fixings are based on a distance metric between the known mixed integer program solution and a relaxation solution. *Rebounding* of a variable means imposing new lower and upper bounds on the variable by changing its current bounds.

In [29], Danna et al. tried two hybrid strategies of RINS and LB and concluded that the resulting performances were not better than that of RINS alone. In this thesis, we show that DINS outperforms both RINS and LB on a benchmark test suite that includes all the instances from Danna et al. [29] as well as many other instances.

Like RINS, DINS also relies on the fact that, during the mixed integer program search tree exploration, the relaxation solutions at those nodes that are not pruned always provide a better objective value than that of the current mixed integer program solution.

But unlike in RINS, the intuition in DINS is that the improved mixed integer program solutions are more likely to be close to the current relaxation solution. An exact modeling of this intuition would require the inclusion of the following quadratic inequality,

$$\sum_{j \in N} (x_j - x_{j(node)})^2 \leq \sum_{j \in N} (x_{j(mip)} - x_{j(node)})^2,$$

where x_{mip} and x_{node} denote the current mixed integer program solution and the current relaxation solution, and for a variable x_j, $x_{j(mip)}$ and $x_{j(node)}$ denote the values of x_j in x_{mip} and x_{node} respectively.

Unfortunately, this quadratic inequality cannot be expressed as a linear programming constraint. DINS relaxes the intuition by considering that the improved solutions are close to x_{node} only with respect to the integer-constrained variables, as measured by the following inequality based on the absolute differences.

$$\sum_{j \in B \cup G} |x_j - x_{j(node)}| \leq \sum_{j \in B \cup G} |x_{j(mip)} - x_{j(node)}|.$$

DINS then partially captures this inequality, the chosen distance metric, by defining a neighbourhood with some rebounding, some hard fixing, and some soft fixing of the integer-constrained variables.

The details of its neighbourhood definition are as follows.

Notice that if an integer-constrained variable, for which the absolute difference between $x_{j(mip)}$ and $x_{j(node)}$ is less than half, takes a different value than $x_{j(mip)}$ in an improved solution, the absolute difference increases. For example, assume that the lower and upper bound of an integer-constrained variable x_j are 0 and 3 respectively. Also assume that $x_{j(mip)} = 2$ and $x_{j(node)} = 1.7$. Then the absolute difference is 0.3 which is less than half. Now if x_j takes any integer values from [0, 3] other than 2, the absolute difference will be greater than 0.3. On the other hand, if an integer-constrained variable, for which the absolute difference between $x_{j(mip)}$ and $x_{j(node)}$ is greater or equal to half, takes a different value than $x_{j(mip)}$ in an improved solution, the absolute difference may not increase. For example, assume that for the same variable x_j, $x_{j(mip)} = 2$ and $x_{j(node)} = 1.3$. Then the absolute difference is 0.7 which is greater than half. Now if x_j takes the value 1, the absolute difference decreases; if it takes value 0 or 3, the absolute difference increases.

DINS changes the lower and upper bounds of an integer-constrained variable x_j, for which the absolute difference between its value in x_{mip} and x_{node} is greater or equal to half, so that at an improved solution this absolute difference does not increase. Considering l_j^{old} and u_j^{old} as the

existing lower and upper bounds of x_j, DINS computes the new lower and upper bound l_j^{new} and u_j^{new} respectively in the following way.

if $(x_{j(mip)} \geq x_{j(node)})$ then

$$l_j^{new} \leftarrow \max(l_j^{old}, \lceil x_{j(node)} - (x_{j(mip)} - x_{j(node)}) \rceil), \quad u_j^{new} \leftarrow x_{j(mip)}$$

elsif $(x_{j(mip)} < x_{j(node)})$ then

$$l_j^{new} \leftarrow x_{j(mip)}, \quad u_j^{new} \leftarrow \min(u_j^{old}, \lfloor x_{j(node)} + (x_{j(node)} - x_{j(mip)}) \rfloor).$$

We call this process rebounding. Rebounding does not change existing bounds for all the integer-constrained variables x_j for which $|x_{j(mip)} - x_{j(node)}| \geq 0.5$. For example, no binary variable, for which $|x_{j(mip)} - x_{j(node)}| \geq 0.5$, changes its bounds.

Now, if all the integer-constrained variables, for which $|x_{j(mip)} - x_{j(node)}| < 0.5$, are fixed to $x_{j(mip)}$, then any solution found from the neighbourhood obtained by rebounding is obviously a closer one to x_{node} in terms of the chosen distance metric.

But, the sum of absolute differences can also decrease, if the total decrease d in the sum of absolute differences caused by the integer-constrained variables for which $|x_{j(mip)} - x_{j(node)}| \geq 0.5$ is greater than the total increase d' in the sum of absolute differences caused by the integer-constrained variables for which $|x_{j(mip)} - x_{j(node)}| < 0.5$. The expression of d and d' are as follows.

$$d = \sum_{\substack{j \in \mathcal{B} \cup \mathcal{G} \wedge \\ |x_{j(mip)} - x_{j(node)}| \geq 0.5}} |x_{j(mip)} - x_{j(node)}| - \sum_{\substack{j \in \mathcal{B} \cup \mathcal{G} \wedge \\ |x_{j(mip)} - x_{j(node)}| \geq 0.5}} |x_j - x_{j(node)}|,$$

and

$$d' = \sum_{\substack{j \in \mathcal{B} \cup \mathcal{G} \wedge \\ |x_{j(mip)} - x_{j(node)}| < 0.5}} |x_j - x_{j(node)}| - \sum_{\substack{j \in \mathcal{B} \cup \mathcal{G} \wedge \\ |x_{j(mip)} - x_{j(node)}| < 0.5}} |x_{j(mip)} - x_{j(node)}|.$$

DINS partially captures this observation by allowing the integer-constrained variables x_j, for which $|x_{j(mip)} - x_{j(node)}| < 0.5$, to change their values in x_{mip} so that d' is not larger than a chosen small

number p. It does this by performing some soft fixing and some hard fixing of these variables. It performs soft fixing through the LB inequality, and as noted in § 2.2.11, the LB inequality requires inclusion of new variables when general integer variables are considered. As in [35] and [29], DINS constructs the LB inequality using only 0-1 variables. Therefore, it fixes all the general integer variables x_j with $|x_{j(mip)} - x_{j(node)}| < 0.5$ at $x_{j(mip)}$. Such fixing is known as hard fixing of variables.

Among the binary variables, for which $|x_{j(mip)} - x_{j(node)}| < 0.5$, DINS performs some hard fixing like RINS, but incorporates some more intuitions in this process. Like RINS, DINS chooses the same set of variables, that agree in both x_{mip} and x_{node}, as the primary candidates for hard fixing. However, we think that all the variables in this primary candidate set are not equally likely to stay in their current values at x_{mip}. Notice that the objective value corresponding to the root relaxation solution, x_{root}, of the search tree provides a lower bound on the objective value of mixed integer program optimal solution at the beginning of search tree, and the previously encountered mixed integer program solutions at a point of execution are the known feasible solutions. Since the objective of DINS is to find improved feasible solutions, from the improvement point of view it uses x_{root}, and from the feasibility point of view it uses the encountered mixed integer program solutions in guiding the hard fixing of binary variables. For this purpose, DINS applies a filtering step to the primary candidate set using two pieces of information. The first of these comes from the intuition that a variable in the primary candidate set, that takes the same value in x_{root} and x_{node}, is more likely to take the same value in improved solutions. The second comes from the intuition that a variable in the primary candidate set, that takes the same value in the previously encountered mixed integer program solutions, is more likely to take the same value in improved solutions. DINS uses an array of flags to keep track of the variables that take different values in the previously encountered mixed integer program solutions. Thus, more explicitly, DINS performs the hard fixing of binary variables in the following way. Let Δ be an array where $\Delta[j]$ is set if x_j takes different values in previously encountered mixed integer program solutions. Then, DINS fixes a binary variable x_j at value $x_{j(mip)}$ if $x_{j(mip)} = x_{j(node)} = x_{j(root)}$ and $\Delta[j]$ is clear.

Let $\mathcal{F}$ and $\mathcal{H}$ denote the set of variables for which rebounding and hard fixing have been performed respectively. Now assume $\mathcal{R}$ is the set of variables where $\mathcal{R} = (\mathcal{B} \cup \mathcal{G}) - \mathcal{F} - \mathcal{H}$. According to our construction $\mathcal{R}$ contains only binary variables. DINS performs soft fixing on the variables in $\mathcal{R}$ by adding the following LB inequality.

$$\sum_{j \in \mathcal{R} \wedge x_{j(mip)}=0} x_j + \sum_{j \in \mathcal{R} \wedge x_{j(mip)}=1} (1 - x_j) \leq p.$$

DINS generates its search neighbourhood taking small values for p. Therefore, a solution found by searching this neighbourhood can have a sum of absolute differences increased by at most p.

The definition of DINS search neighbourhood ends here.

Procedure DINS_at_tree_node
INPUT: a 0-1 mixed integer problem P, the current mixed integer program solution x_{mip}, the current node relaxation solution x_{node}, the root relaxation solution x_{root}, parameter p, node limit nl, and the flag array Δ.
OUTPUT: if successful, return a new mixed integer program solution x^*, otherwise return x_{mip}.

1. if (x_{mip} is a new mixed integer program solution compared to the solution at the termination of last call of this procedure) update the array Δ accordingly.
2. $x^* \leftarrow x_{mip}$; $p_{current} \leftarrow p$; exploredAndNoSolution $\leftarrow$false.
3. repeat
4. construct $P+$ from P as follows:
   (i) perform rebounding of the variables x_j for which $|x_j^* - x_{j(node)}| \geq 0.5$,
   (ii) perform hard fixing of the general integer variables x_j for which $|x_j^* - x_{j(node)}| < 0.5$,
   (iii) perform hard fixing of the binary variables x_j for which $x_j^* = x_{j(node)} = x_{j(root)}$ and $\Delta[j]$ is clear,
   (iv) let $\mathcal{R}$ be the set of remaining binary variables.
   if ($|\mathcal{R}| > p_{current}$) perform soft fixing by adding the inequality $\sum_{j \in \mathcal{R} \wedge x_j^*=0} x_j + \sum_{j \in \mathcal{R} \wedge x_j^*=1} (1 - x_j) \leq p_{current}$.
5. Apply a branch-and-bound or branch-and-cut like exact solver to $P+$ with node limit nl and an objective cutoff equal to the objective value provided by x^*.
6. if (a new solution x_{new} is obtained) then
7. $x^* \leftarrow x_{new}$; $p_{current} \leftarrow p$; update the array Δ.
8. else if (node limit reached without having a new solution) then
9. if ($|\mathcal{R}| = \phi$) $p_{current} \leftarrow -1$.
10. else $p_{current} \leftarrow p_{current} - 5$.
11. else exploredAndNoSolution $\leftarrow$true.
12. until ($p_{current} < 0$ or exploredAndNoSolution)
13. return x^*.

Figure 4.1: A pseudo-code description of DINS.

Now, whenever we apply DINS procedure at a particular node of the mixed integer program search tree, it creates the described neighbourhood with the initial chosen value of p and explores

it using either a branch-and-bound or a branch-and-cut solver with a specified node limit nl. If the exploration reaches the node limit without finding a new solution, as a step to intensify the search, DINS reduces p by 5 to reduce the size of neighbourhood and explores the new neighbourhood. This continues until $p < 0$, or the neighbourhood exploration finds a new solution, or DINS explores the neighbourhood completely without finding a new solution. Whenever the neighbourhood exploration finds a new solution, DINS resets p to its initial chosen value and continues in the same fashion. Figure 4.1 shows the operation sequence of DINS at a particular node of the mixed integer program search tree. At the termination of the procedure, execution returns to the exploration of the mixed integer program search tree. If the procedure finds a new mixed integer program solution, the algorithm updates the mixed integer program solution at the mixed integer program search tree.

Like RINS, we call the DINS procedure first at the node at which the mixed integer program search tree finds the first feasible solution. Thereafter, we call the DINS procedure at every f nodes of mixed integer program search tree for some reasonably large f.

4.2 DINS Performance Evaluation

To evaluate the performance of DINS, we compared it against three solvers: RINS, the recent improvement heuristic, LB, the heuristic from which PGC uses some ideas, and Cplex Version 9.13, a commercial mixed integer programming solver, in its default setup called Cplex-D.

From the benchmark suite shown in Table B.1- B.3 of Appendix B, we chose all 64 mixed integer program instances with the exception of those instances for which Cplex-D either gives the proof of optimality or fails to find a solution in one CPU-hour. We also chose some pseudo-randomly generated instances of two different class of problems. One class is from the constrained market-sharing problem defined in Chapter 6. The other is from the Cornuéjols-Dawande optimality model of market-sharing problem described in § 2.3.2.

We ran all experiments on a 2403 MHz AMD Athlon processor with 128 MByte of memory under Redhat Linux 9.0. We implemented LB, RINS, and DINS in the C programming language with the mixed integer program search tree generated by Cplex-D. We alloted one CPU-hour to each

solver, and it seemed to be sufficient to distinguish the effectiveness of all the solvers.

The three solvers namely LB, RINS, and DINS have a set of parameters which needed to be set. As in [29], for LB we set $p = 10$ and $nl = 1000$, and for RINS we used Cplex 9.13 with the parameter IloCplex::MIPEmphasis set to 4, where according to [29] $f = 100$ and $nl = 1000$. For DINS we set $p = 5$, different from LB to allow the neighbourhood to voilate the chosen distance metric little and to keep the neighbourhood small, $f = 100$, and $nl = 1000$ as in RINS.

There is no exact way to distinguish a good and a poor mixed integer program solution without knowing the optimal solution. However, following Danna et al. [29] we presume that first mixed integer program solution found by Cplex-D represents a poor solution, and the mixed integer program solution found by Cplex-D in one CPU-hour represents a good solution. In our experiment, we attempted to see how DINS compare against RINS, LB, and Cplex-D starting from both the presumed poor and good initial solution on the chosen benchmark instances.

4.2.1 DINS Performance Evaluation from the Presumably Poor Solutions

We first invoked the Cplex-D to find the first solution and then invoked different solvers with the found solution as a known solution at the root node of the mixed integer program search tree. This provided all four solvers the same starting solution.

In order to capture the quality of obtained solution by each solver, we use the measure *percentage of gap* defined by 100*|(objective value of obtained solution − objective value of the best known solution)/objective value of the best known solution|.

Table B.13 of Appendix B shows the percentage of gap obtained at the end of one CPU-hour by all the four solvers, where the bold face identifies the best solver for the corresponding instance; multiple bold faces appear for an instance if there are multiple solvers obtaining the same solution.

Following Danna et al. [29], we group the instances into three different sets so that the effectiveness of different solvers in different groups becomes visible. According to [29], an instance is in group 'small spread', 'medium spread', and 'large spread', if the gap between its worst solution, among the found solutions by four solvers, and its best known solution is less than 10%, between

10% and 100%, and larger than 100% respectively. The percentage of gaps shown in Table B.13 are used to group the instances.

We use three measures to evaluate the performance of different solvers.

Our first measure is the number of instances for which a solver finds a better solution than the solution obtained by other solvers.

Our second measure is the percentage of gap. We calculate the average and the standard deviation of percentage of gaps obtained on a group of instances.

Our third measure is the *percentage of improvement* defined by 100*|(objective value of the initial solution − objective value of the obtained solution) /objective value of the initial solution|. We calculate the average and the standard deviation of percentage of improvements obtained on a group of instances.

Table 4.1- 4.3 summarize the performance of DINS, with respect to the first measure, against Cplex-D, LB, and RINS respectively. Table 4.4 and 4.5 show the average and the standard deviation of other two measures respectively for different solvers.

Table 4.1: Cplex-D versus DINS starting from a presumably poor solution on 64 benchmark instances. Cplex-D better: the number of instances at which Cplex-D finds better solution than DINS. DINS better: the number of instances at which DINS finds better solution than Cplex-D. Tied: the number of instances at which both Cplex-D and DINS find the same improved solution. Entries indicate number of instances.

Cplex-D better	DINS better	Tied
10	44	10

Table 4.2: LB versus DINS starting from a presumably poor solution on 64 benchmark instances. Entries indicate number of instances.

LB better	DINS better	Tied
8	49	7

Table 4.3: RINS versus DINS starting from a presumably poor solution on 64 benchmark instances. Entries indicate number of instances.

RINS better	DINS better	Tied
20	32	12

Table 4.4: The average ($\overline{x}$) and the standard deviation (δ) of percentage of gaps obtained by Cplex-D, LB, RINS, and DINS starting from a presumably poor solution on 64 benchmark instances.

	Cplex-D		LB		RINS		DINS	
	$\overline{x}$	δ	$\overline{x}$	δ	$\overline{x}$	δ	$\overline{x}$	δ
On all 64 instances	44.22	191.29	51.19	188.84	41.33	253.77	39.73	232.19
On 45 small spread instances	1.86	2.35	1.81	2.44	1.05	1.99	0.97	1.72
On 13 medium spread instances	15.41	12.74	17.72	17.25	10.35	13.94	9.74	12.64
On 6 large spread instances	424.28	518.63	494.07	433.95	410.51	793.91	395.38	715.03

Table 4.5: The average ($\overline{x}$) and the standard deviation (δ) of percentage of improvements obtained by Cplex-D, LB, RINS, and DINS starting from a presumably poor solution on 64 benchmark instances.

	Cplex-D		LB		RINS		DINS	
	$\overline{x}$	δ	$\overline{x}$	δ	$\overline{x}$	δ	$\overline{x}$	δ
On all 64 instances	36.19	35.08	35.49	34.01	38.01	35.49	38.05	35.43
On 45 small spread instances	23.41	29.17	23.61	29.02	23.90	29.22	23.92	29.14
On 13 medium spread instances	60.43	27.93	60.25	27.34	62.05	25.61	62.29	25.44
On 6 large spread instances	80.78	28.96	70.90	31.60	91.64	6.31	91.66	6.22

The results shown in Table 4.1- 4.5 suggest that starting from the presumably poor solutions, DINS is better than the other three solvers with respect to all three measures.

Now, for different group of instances, Figure 4.2- 4.4 show how solution quality, the average percentage of gap, changes over time for different solvers. Analyzing these figures, we find the following differences among the solvers in these three group of instances. For all three group of instances, DINS is worse than RINS at the initial level of computation, but it becomes better as the computation progresses, and once it becomes better, it maintains its lead over RINS for the remaining part of the computation. For small and large spread instances, DINS obtains the lead over RINS earlier than in medium spread instances. Similarly in medium and large spread instances, DINS is worse than Cplex-D at the initial level of computation, but it becomes better as the computation progresses. LB is always worse than RINS and DINS where, at the end of time limit, LB has an edge over Cplex-D only in small spread instances.

4.2.2 DINS Performance Evaluation from the Presumably Good Solutions

We first invoked the Cplex-D for one CPU-hour and then invoked different solvers with the found solution from the Cplex-D as a known solution at the root node of the mixed integer program search tree. This provided all four solvers the same presumably good starting solution.

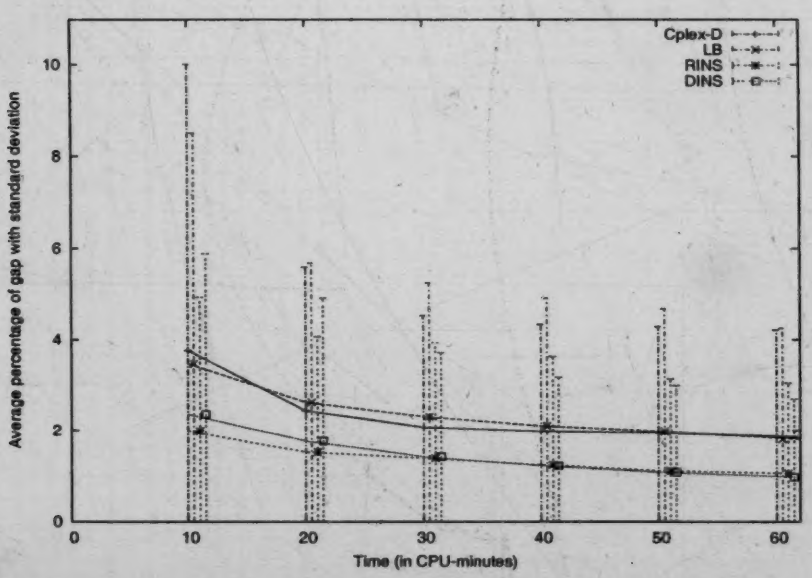

Figure 4.2: Curves in horizontal direction show the change in average percentage of gap by different solvers on the small spread instances starting from presumably poor solution. Vertical lines show the standard deviation from the average.

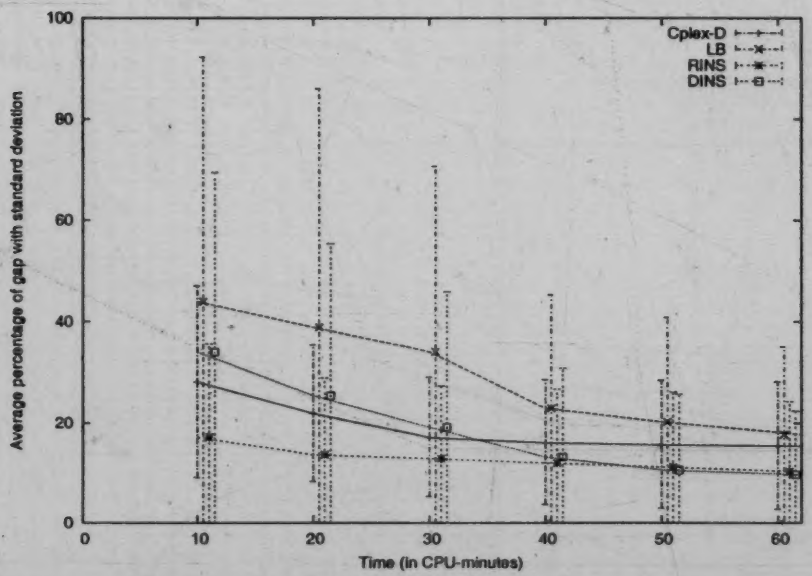

Figure 4.3: Curves in horizontal direction show the change in average percentage of gap by different solvers on the medium spread instances starting from presumably poor solution. Vertical lines show the standard deviation from the average.

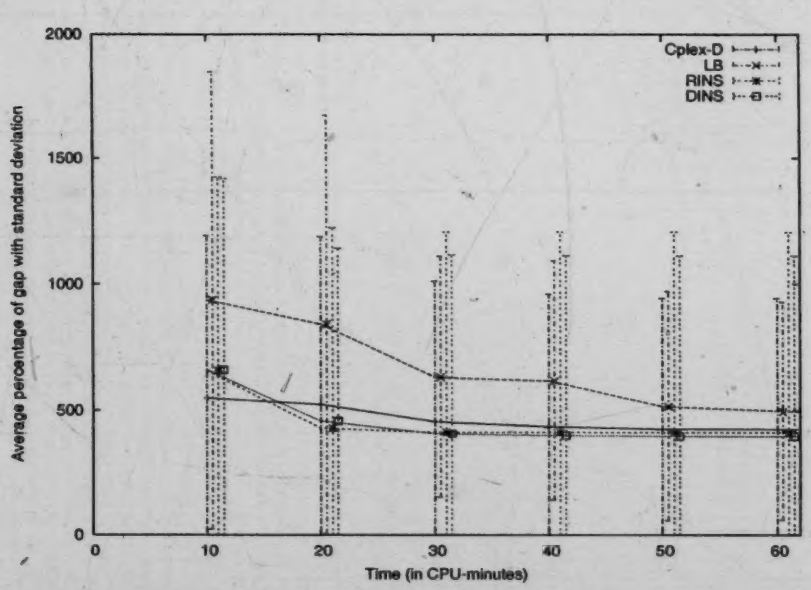

Figure 4.4: Curves in horizontal direction show the change in average percentage of gap by different solvers on the large spread instances starting from presumably poor solution. Vertical lines show the standard deviation from the average.

Table 4.6- 4.8 summarize the performance of DINS, with respect to the first measure, against Cplex-D, LB, and RINS respectively. Table 4.9 and 4.10 show the average and the standard deviation of other two measures respectively for different solvers.

Table 4.6: Cplex-D versus DINS starting from a presumably good solution on 64 benchmark instances. Cplex-D better: the number of instances at which Cplex-D finds better solution than DINS. DINS better: the number of instances at which DINS finds better solution than Cplex-D. Tied: the number of instances at which both Cplex-D and DINS find the same improved solution. No new solution: the number of instances at which both Cplex-D and DINS fail to find a new solution. Entries indicate number of instances.

Cplex-D better	DINS better	Tied	No new solution
7	43	2	12

Table 4.7: LB versus DINS starting from a presumably good solution on 64 benchmark instances. Entries indicate number of instances.

LB better	DINS better	Tied	No new solution
9	35	9	11

Table 4.8: RINS versus DINS starting from a presumably good solution on 64 benchmark instances. Entries indicate number of instances.

RINS better	DINS better	Tied	No new solution
11	30	12	11

Table 4.9: The average ($\overline{x}$) and the standard deviation (δ) of percentage of gaps obtained by Cplex-D, LB, RINS, and DINS starting from a presumably good solution on 64 benchmark instances.

	Cplex-D		LB		RINS		DINS	
	$\overline{x}$	δ	$\overline{x}$	δ	$\overline{x}$	δ	$\overline{x}$	δ
On all 64 instances	32.43	172.67	31.67	172.77	31.21	172.82	29.14	161.29
On 45 small spread instances	1.41	1.92	1.07	1.79	0.56	0.92	0.54	1.17
On 13 medium spread instances	13.57	14.67	10.63	13.45	10.46	13.00	8.59	11.48
On 6 large spread instances	305.92	525.17	306.77	524.64	306.06	525.07	288.17	488.36

Table 4.10: The average ($\overline{x}$) and the standard deviation (δ) of percentage of improvements obtained by Cplex-D, LB, RINS, and DINS starting from a presumably good solution on 64 benchmark instances.

	Cplex-D		LB		RINS		DINS	
	$\overline{x}$	δ	$\overline{x}$	δ	$\overline{x}$	δ	$\overline{x}$	δ
On all 64 instances	2.35	11.28	3.04	11.31	3.45	11.23	3.96	11.38
On 45 small spread instances	0.45	0.98	0.78	1.26	1.26	1.97	1.29	1.88
On 13 medium spread instances	2.50	4.43	4.91	5.23	5.10	3.50	6.57	4.73
On 6 large spread instances	16.31	35.78	15.96	35.76	16.27	35.76	18.47	34.82

The results shown in Table 4.6- 4.10 suggest that DINS is better than the other three solvers with respect to all three measures starting from the presumably good solutions. Only with respect to standard deviation, DINS seems to be little bit worse than RINS in small spread instances.

Now, for different group of instances, Figure 4.5- 4.7 show how solution quality, the average percentage of gap, changes over time for different solvers starting from the presumably good solutions. Analyzing these figures, we find the following differences among the solvers in these three

group of instances. For small spread instances, DINS is worse than RINS at the initial level of computation and it gets better as computation progresses. For medium spread instances, DINS is better than the other three solvers throughout the computation. For large spread instances, DINS becomes better than RINS as computation progresses. Both Cplex-D and LB are worse comparing to DINS throughout the computation in all three group of instances.

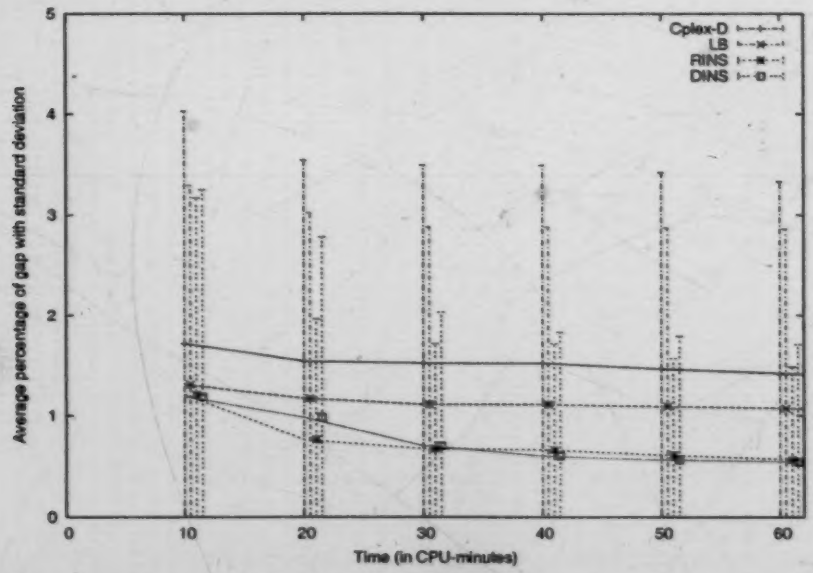

Figure 4.5: Curves in horizontal direction show the change in average percentage of gap by different solvers on the small spread instances starting from presumably good solution. Vertical lines show the standard deviation from the average.

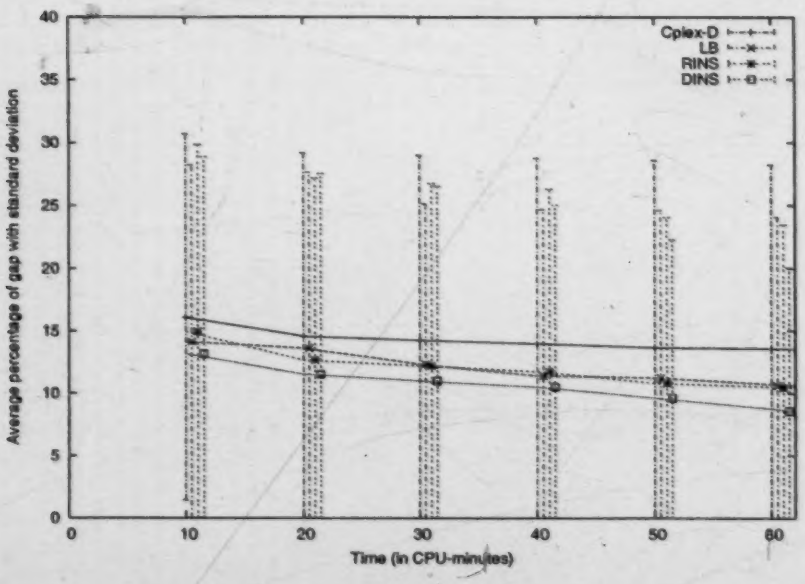

Figure 4.6: Curves in horizontal direction show the change in average percentage of gap by different solvers on the medium spread instances starting from presumably good solution. Vertical lines show the standard deviation from the average.

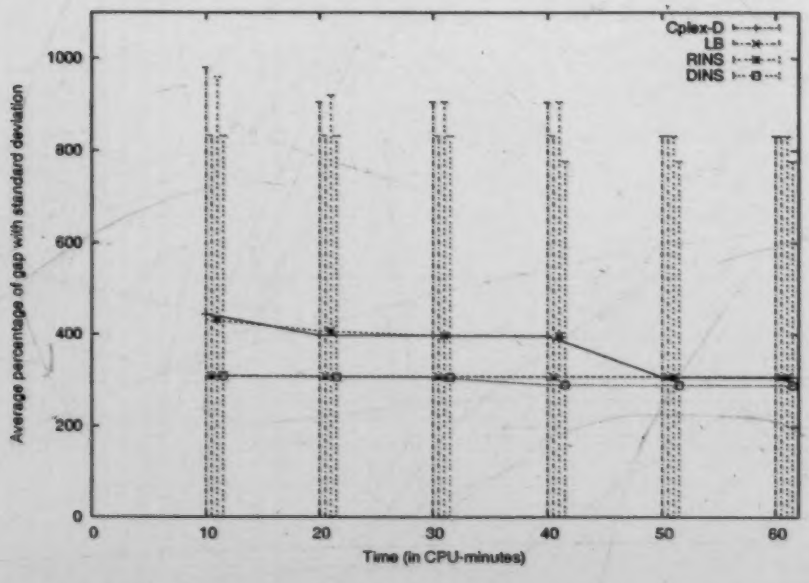

Figure 4.7: Curves in horizontal direction show the change in average percentage of gap by different solvers on the large spread instances starting from presumably good solution. Vertical lines show the standard deviation from the average.

4.2.3 DINS Neighbourhoods versus RINS Neighbourhoods

The experiments suggest that on the benchmark instances, it is more effective to explore the neighbourhoods defined by DINS than that of RINS. Although size of a mixed integer program instance does not reflect the true hardness to solve it, in an attempt to compare the neighbourhoods defined by DINS and RINS, we tried to estimate the enumeration size of both neighbourhoods.

For simplicity, we restricted this analysis only to 0-1 mixed integer program instances.

In RINS, let k denotes the number of neighbourhoods explored in one CPU-hour for a particular instance with n_B binary variables, and f_i, $1 \leq i \leq k$, denotes the number of free binary variables, which may take either value 0 or value 1, at the i-th neighbourhood. Then we define *average enumeration ratio* for RINS as follows:

$$ER := \frac{\left(\sum_{i=1}^{k} \log 2^{f_i}\right)/k}{\log 2^{n_B}} = \frac{\left(\sum_{i=1}^{k} f_i\right)/k}{n_B},$$

where 2^{n_B} is the size of the enumeration space when all the binary variables are allowed to take either value 0 or value 1, and 2^{f_i} is the size of the enumeration space when f_i binary variables are allowed to take either value 0 or value 1 keeping the remaining binary variables fixed at some values. We apply logarithm on the size of the enumeration space to express the enumeration ratio as the ratio of number of free binary variables.

Similarly in DINS, let k_5 and k_0 denote the number of neighbourhoods, with soft fixing parameter p set to 5 and 0 respectively, explored in one CPU-hour for a particular instance with n_B binary variables. Also let f_i and s_i, $1 \leq i \leq k_5$, respectively denote the number of free binary variables and the number of binary variables pertained to soft fixing inequality at the i-th neighbourhood with $p = 5$, and let l_i, $1 \leq i \leq k_0$, denotes the number of free binary variables at the i-th neighbourhood with $p = 0$. Then we define average enumeration ratio for DINS as follows:

$$ER := \frac{\left(\sum_{i=1}^{k_5} \log 2^{(f_i+r_i)} + \sum_{i=1}^{k_0} \log 2^{l_i}\right)/(k_5+k_0)}{\log 2^{n_B}} = \frac{\left(\sum_{i=1}^{k_5}(f_i+r_i) + \sum_{i=1}^{k_0} l_i\right)/(k_5+k_0)}{n_B},$$

where we approximate the enumeration size $\sum_{t=0}^{p}\binom{s_i}{t}$ of soft fixing inequality by the parameter r_i such that $2^{r_i-1} < \sum_{t=0}^{p}\binom{s_i}{t} \leq 2^{r_i}$.

Table B.18 of Appendix B shows the number of explored neighbourhoods and the average enumeration ratio for both RINS and DINS starting from presumably poor solutions. In most of the instances, DINS had a lesser average enumeration ratio than RINS. And in most of the instances for which DINS had a lesser average enumeration ratio than RINS, DINS explored more neighbourhoods than RINS. This observation leads us to conjecture that, in the case where heuristics are applied for a long time, exploring useful small neighbourhoods is better than exploring useful large neighbourhoods, where neighbourhoods are considered useful if a node-limited search on the neighbourhoods can yield improved solutions.

4.2.4 Verification of Intuitions used in DINS

In an attempt to see how good were the intuitions on which we designed DINS, we provide some statistical measures from our experimental results. We found that, the number of times neighbourhood exploration found a new solution in all the instances, the chosen distance metric was satisfied in 80.89% occurrences and the quadratic distance metric was satisfied in 80.5% occurrences. These experimental results support our intuition that improved solutions are more likely to be close to the node relaxation solutions, and also support our choice of distance metric. Moreover, relaxing the chosen distance metric a little bit using the parameter p perhaps gives DINS the extra power of finding those improved solutions which do not satisfy the chosen distance metric at the found node, but probably would satisfy the chosen distance metric at some deeper nodes of the MIP search tree.

Unlike RINS, DINS uses the root relaxation solution and the encountered mixed integer program solutions in guiding the hard fixing of binary variables. Experiments showed that this had an effect in finding the good mixed integer program solutions. We implemented a modified DINS where we performed the hard fixing of binary variables according to the hard fixing suggested in RINS. In an experiment on the 64 benchmark instances starting from the presumably poor solu-

tions, between DINS and modified DINS, DINS found equal or better solution in 45 instances, whereas the modified one found equal or better solution in 34 instances.

4.2.5 Performance on Randomly Generated Instances

We first show the performance of different solvers on a set of pseudo-randomly generated Cornuéjols-Dawande optimality-hard instances described in § 2.3.2.

For instances with n variables and m constraints, Cornuéjols and Dawande showed that picking the relation $n = 10(m-1)$ yielded the hardest instances for their optimality model of market-sharing problem. Using $n = 10(m-1)$, we pseudo-randomly generated 10 Cornuéjols-Dawande optimality-hard instances with 40 to 100 variables each. We used the first solution found by Cplex-D as the starting solution for all four solvers, namely Cplex-D, LB, RINS, and DINS. We alloted one CPU-hour to each solver.

In the experiments with pseudo-randomly generated instances, we do not use the average percentage of gap as a measure, since we have no better information about the best known solutions for this set of instances than the information obtained from our experiments. Table 4.11- 4.14 summarize the performance of DINS against Cplex-D, LB, and RINS respectively on this set of instances. For detailed result, see Table B.14 of Appendix B.

Table 4.11: Cplex-D versus DINS on 70 pseudo-randomly generated Cornuéjols-Dawande optimality-hard instances of different sizes. Entries indicate number of instances.

Cplex-D better	DINS better	Tied
34	24	12

Table 4.12: LB versus DINS on 70 pseudo-randomly generated Cornuéjols-Dawande optimality-hard instances of different sizes. Entries indicate number of instances.

LB better	DINS better	Tied
40	21	9

Table 4.13: RINS versus DINS on 70 pseudo-randomly generated Cornuéjols-Dawande optimality-hard instances of different sizes. Entries indicate number of instances.

RINS better	DINS better	Tied
33	21	16

Table 4.14: The average and the standard deviation of percentage of improvements obtained by Cplex-D, LB, RINS, and DINS on 70 pseudo-randomly generated Cornuéjols-Dawande optimality-hard instances.

	Cplex-D	LB	RINS	DINS
average of percentage of improvements	99.17	99.18	99.15	99.12
standard deviation of percentage of improvements	0.40	0.41	0.41	0.39

The results shown on Table 4.11- 4.14 suggest that DINS is not better than other solvers on these set of instances. However, the performance of DINS is very close to that of other solvers.

Next, we show the performance of different solvers on a set of pseudo-randomly generated constrained market-sharing instances described in Chapter 6.

As in § 3.4.5, we generated instances from three groups. A parameter k differentiates these groups, where k is introduced to relate the number of variables n and the number of constraints m of instances. The relation between n and m is defined by $m = \lfloor \frac{n}{k} \rfloor$. For the first group with $k = 2.0$, the instances generated are of 50, 100, and 150 variables, and for the groups with $k = 1.5$ and 1.3, the instances are of 50, 75, and 100 variables.

Improvement heuristics require a feasible solution to start with. Since PGC_1 performed better on these set of instances, we picked first 10 generated instances of each problem size for which PGC_1 found a feasible solution. We gave the feasible solutions obtained by PGC_1 as the starting feasible solutions to all solvers.

In this experiment, the first measure to evaluate the performance of a solver is the number of instances for which a solver finds a new solution. We do not use this measure in the earlier experiments since the solvers do not show much difference with respect to this measure. Table 4.15- 4.18 summarize the performance of DINS against Cplex-D, LB, and RINS respectively on this set of instances. For detailed result, see Table B.15- B.17 of Appendix B.

Table 4.15: Cplex-D versus DINS on 90 pseudo-randomly generated constrained market-sharing instances of different sizes. Entries indicate number of instances.

Cplex-D better	DINS better	No new solution
0	44	46

Table 4.16: LB versus DINS on 90 pseudo-randomly generated constrained market-sharing instances of different sizes. Entries indicate number of instances.

LB better	DINS better	Tied	No new solution
0	43	1	46

Table 4.17: RINS versus DINS on 90 pseudo-randomly generated constrained market-sharing instances of different sizes. Entries indicate number of instances.

RINS better	DINS better	Tied	No new solution
5	38	1	46

Table 4.18: The average and the standard deviation of percentage of improvements obtained by Cplex-D, LB, RINS, and DINS on 90 pseudo-randomly generated constrained market-sharing instances.

	Cplex-D	LB	RINS	DINS
average of percentage of improvements	0.00	0.99	8.01	13.43
standard deviation of percentage of improvements	0.00	4.92	17.11	17.93

The results shown on Table 4.15- 4.18 suggest that DINS performs better than other solvers on these pseudo-randomly generated constrained market-sharing instances.

Chapter 5

Neighbourhood Pivot and Gomory Cut

In this chapter we present NEIGHBOURHOOD PIVOT AND GOMORY CUT (NPGC), a new 'find-and-improve' type mixed integer programming heuristic.

NPGC is similar in approach to the Dana et al.'s implementation of LB heuristic. Recall from § 2.2.11 that in Danna et al.'s implementation of LB, the main procedure calls the LB procedure when the search tree, generated by either a branch-and-bound or a branch-and-cut solver, finds the first feasible solution. Thereafter, as a process of diversification, the main procedure calls the LB procedure whenever the search tree obtains a new feasible solution.

NPGC, essentially an extension of PGC, defines a Gomory Cut based search neighbourhood and explores it using either a branch-and-bound or a branch-and-cut solver with a node limit. If the exploration of a search neighbourhood provides a feasible solution, NPGC calls the LB procedure with the feasible solution. Otherwise, NPGC defines a new search neighbourhood and continues accordingly. NPGC calls the LB procedure each time NPGC finds a new feasible solution. Notice that NPGC actually replaces the search tree used by Danna et al. by its Gomory cut based neighbourhood search.

NPGC is also similar to the PS[2004] of Balas et al. [16]. As noted in § 2.2.2, as is the case

with NPGC, PS[2004] also performs a neighbourhood search around the dead-end reached after a sequence of pivoting. There are two major differences between NPGC and PS[2004]. First, NPGC defines the neighbourhood using Gomory cuts, whereas PS[2004] defines its neighbourhood using inequalities based on values of integer variables at the dead-end. Second, in NPGC execution begins on a new search neighbourhood whenever exploration of the current search neighbourhood fails to find a solution or whenever the improvement procedure terminates; by contrast, PS[2004] calls the Xpress mixed integer program solver whenever exploration of the current search neighbourhood fails. As reported in [16], an initial implementation of PS[2004] was weaker in performance than the final implementation, which ran on the commercial mixed integer program solver Xpress Version 14.2. Furthermore, the final implementation used some platform-dependent time settings. In particular, within the initial search phase, a total of 5 seconds was alloted for pivoting; within each improvement, a total of 30 seconds was alloted for shifting. In light of the fact that NPGC turned out to be less effective than the recent heuristic RINS, the significance of determining the relative strength of NPGC versus PS[2004] is somewhat moot.

In the next section we see the details of NPGC.

5.1 Neighbourhood Pivot and Gomory Cut

Recall from § 3.4.6 that for some benchmark instances, the initial search phase of PGC runs out of time before finding a feasible solution or even reaching a dead end. This motivates the addition of a neighbourhood search to the PGC framework. We use the same notation that we have defined in PGC.

NPGC applies the search phase of PGC constituting only Type 1 pivots, since it ensures the termination of the search phase after a finite number of steps. At the end of the PGC search phase, if the current basic feasible solution, x^*, of $\mathcal{L}(P^+)$ is P-feasible, NPGC puts an upper bound on the objective value by adding $cx \leq cx^*$ to P and calls the LB procedure LB_at_tree_node shown in Figure 2.7 to improve the obtained P-feasible solution. Otherwise, using the intuition that a P-feasible solution will be somewhere around x^*, NPGC defines a search neighbourhood around

x^* using Gomory cuts and explores that by either a branch-and-bound or a branch-and-cut mixed integer program solver.

To define the aforementioned search neighbourhood, NPGC generates a Gomory cut from the tableau row corresponding to the most integer-infeasible binary variable and expresses it as $\alpha x \geq \beta$ in terms of the decision variables. NPGC defines the neighbourhood by adding to P the Gomory cut $\alpha x \geq \beta$ and another inequality $\alpha x \leq \beta + d$ representing a parallel hyperplane to the Gomory cut. The parameter d is chosen so that the orthogonal distance between $\alpha x = \beta$ and $\alpha x = \beta + d$ is $\frac{\sqrt{n_c}}{k}$, where k is an execution parameter, and n_c is the number of variables in the cut $\alpha x \geq \beta$. Notice that, for 0-1 programs, since the largest diagonal length of 0-1 polytope in the space of n_c variables is $\sqrt{n_c}$, choosing $k \leq 1$ causes the search neighbourhood polytope to include the complete feasible region of the P-polytope.

NPGC applies either a branch-and-bound or a branch-and-cut mixed integer program solver on the search neighbourhood with a specified node limit nl_s. When the solver terminates from a neighrbourhood search, it removes the inequalities $\beta \leq \alpha x \leq \beta + d$ from P and then modifies P based on the termination statuses in the following way. If the solver completely explores the neighbourhood, namely if the solver finds an optimal solution or it proves the neighbourhood to be integer infeasible, then NPGC adds the inequality $\alpha x \geq \beta + d$ to P. If the solver reaches the node limit nl_s during the search, NPGC adds only the cut $\alpha x \geq \beta$ to P. And, whenever the mixed integer program solver returns a P-feasible solution x^*_{new}, NPGC puts an upper bound on the objective value by adding $cx \leq cx^*_{new}$ to P.

If the solver terminates from the search neighbourhood with a P-feasible solution, NPGC calls the LB procedure LB_at_tree_node shown in Figure 2.7. Whenever LB obtains a new P-feasible solution x^*_{new}, NPGC puts an upper bound on the objective value by adding $cx \leq cx^*_{new}$ to P.

Either the mixed integer program solver terminates from the search neighbourhood without a P-feasible solution or LB terminates from its call, assuming P denotes the modified P, NPGC transforms P to P^+, re-optimizes the resulting $\mathcal{L}(P^+)$, and restarts execution at the initial PGC search phase.

Figure 5.1 shows the algorithm NPGC.

We designed NPGC as a heuristic. However, it can be implemented as an exact solver. If we set the node limit nl_s of the solver applied on search neighbourhood to ∞, the solver explores every search neighbourhood completely. Since $n_c \geq 1$, every search neighbourhood contains a certain portion of the P-polytope for a certain value of k. Considering P as a bounded and finite dimensional polytope, we can show that the P-polytope constitutes of a finite number of search neighbourhood, and thus the complete exploration of all the search neighbourhoods makes the complete exploration of the P-polytope. We limit our analysis for the 0-1 integer programs. The analysis takes the same course for arbitrary integer programs. For a 0-1 program with n variables, we establish a upper-bound k^n on the number of search neighbourhoods. For any αx, $\alpha x \leq \frac{\sqrt{n_c}}{k}$ includes a small n-dimensional hypercube with an edge of length at least $\frac{1}{k}$ in each direction from the current basic feasible solution x^*. Since the big n-dimensional 0-1 hypercube, namely the hypercube with an edge of unit distance in each direction from the origin, holds any given 0-1 polytope and constitutes of k^n small n-dimensional hypercube mentioned earlier, the maximum number of search neighbourhood for 0-1 program is k^n.

5.2 NPGC Performance Evaluation

To evaluate the performance of NPGC, we compared it against three solvers: RINS, the recent improvement heuristic, LB, the heuristic which NPGC uses, and Cplex Version 9.13, a commerical mixed integer programming solver, in its default setup referenced as Cplex-D.

From the benchmark suite shown in Table B.1- B.3, we chose all 53 0-1 mixed integer program instances with the exception of those instances for which Cplex-D gives the proof of optimality in one CPU-hour.

We ran all experiments on a 2403 MHz AMD Athlon processor with 128 MByte of memory under Redhat Linux 9.0. We implemented NPGC in the C programming language where the PGC search phase was designed on top of the version 4.7 of the open source GLPK, and the search neighbourhoods were explored by the Cplex-D. Similarly we implemented LB and RINS in the C

Algorithm NPGC

INPUT: a 0-1 mixed integer problem P, a search node limit nl_s, the parameter p and nl for procedure LB_at_tree_node, a global time limit T and the parameter k.

OUTPUT: A P-feasible solution S_{MIP} (null in case of failure).

```
1.  S_MIP ←null;  elapsedTime ← 0.
2.  repeat
3.      construct L(P+) from P.
4.      solve L(P+) using the bounded variable revised simplex algorithm.
5.      if (L(P+) is infeasible) then return S_MIP.
6.      else S_LP ← the optimal solution found by solving L(P+).
7.      isFeas ← (S_LP is P-feasible).
8.      if (not isFeas) then
9.          construct the Gomory Cut αx ≥ β
                from the row corresponding to the most integer-infeasible binary variable.
10.         while (not isFeas and a PGC1 Type 1 pivot exists)
11.             perform PGC1 Type 1 Pivot;  S_LP ← resulting L(P+)-feasible solution.
12.             if (S_LP satisfies αx ≥ β) then
13.                 isFeas ← (S_LP is P-feasible).
14.                 if(not isFeas) then
15.                     add αx ≥ β in L(P+).
16                      construct the resulting new Gomory Cut αx ≥ β from the row
                          corresponding to the most integer-infeasible binary variable.
17.     if (not isFeas) then
18.         construct the Gomory cut αx ≥ β based on the current L(P+)
                from the row corresponding to the most integer-infeasible binary variable.
19.         determine the parameter d using k.
20.         construct a search neighbourhood by adding αx ≥ β and αx ≤ β + d to P.
21.         apply either a branch-and-bound or a branch-and-cut mixed integer program
                solver on the search neighbourhood with node limit nl_s.
22.         modify P and nl_s based on the termination statuses of the MIP-solver.
23.         if (feasible solution found by the MIP-solver) then
24.             isFeas ←true.
25.     if(isFeas)
26.         update S_MIP and add the objective cutoff to P.
27.         apply procedure LB_at_tree_node with parameter p and nl.
28.         if (the procedure LB_at_tree_node finds a new P-feasible solution)
29.             update S_MIP and add the objective cutoff to P.
30. until (elapsedTime≥ T)
31. return S_MIP.
```

Figure 5.1: A pseudo-code description of NPGC.

programming language with the mixed integer program search tree generated by Cplex-D. Notice that within the implementation of LB, we had to implement Procedure LB_at_tree_node which we used in NPGC as an improvement procedure. Also notice that LB and RINS are improvement heuristics, and so require an initial solution to start with. Since LB and RINS were implemented within the search tree generated by Cplex-D, they started working when Cplex-D found a solution.

The three methods namely LB, RINS, and NPGC have a set of parameters which needed to be set. As in [29], for LB we set $p = 10$ and $nl = 1000$, and for RINS we used Cplex 9.13 with the parameter IloCplex::MIPEmphasis set to 4, where according to [29] $f = 100$ and $nl = 1000$.

Since, in NPGC, it is better to obtain the feasible solution as early as possible, we ran an experiment on choosing parameter k. For that experiment, as is the case with LB and RINS, we set node limit $nl_s = 1000$ in NPGC. Then we ran NPGC for one CPU-hour on the 53 benchmark instances with $k = 1, 10, 100$, and 1000. NPGC failed to find solutions for 13, 13, 8, and 18 instances with $k = 1, 10, 100$, and 1000 respectively. It suggests that if the size of neighbourhood is too large, as in k=1 or 10, the solver is not able to find a solution earlier. On the other hand, if the size of neighbourhood is too small, as in k=1000, the neighbourhood may not include a solution. Following this experimental results, we set $k = 100$. Since our goal was to design NPGC as a heuristic rather than an exact solver, we set $nl_s = 1000$ instead of ∞, and whenever the solver became unsuccessful in finding a feasible solution within the node limit, in an attempt to explore the next neighbourhoods more intensely, we doubled the nl_s. We alloted one CPU-hour to each solver.

We use the measure 'percentage of gap' as defined in § 4.2.1 to capture the quality of obtained solutions. Table B.19 of Appendix B shows the percentage of gap obtained at the end of one CPU-hour by all the four solvers, where the bold face identifies the best solver for the corresponding instance; multiple bold faces appear for an instance if there are multiple solvers obtaining the same solution.

From the results shown on Table B.19 of Appendix B, we see that all the solvers find solutions in 47 out of 53 benchmark instances. Notice that, NPGC fails to find a solution for two instances, namely 'net12' and 'dc11', for which Cplex-D finds solutions. On the other hand, Cplex-D fails to

find a solution for two instances, namely 'protfold' and 'rd-rplusc-21', for which NPGC finds solutions. Since LB and RINS start working when Cplex-D finds the first solution, they are successful in the same number of instances as Cplex-D.

Following the criteria shown in § 4.2.1 we group the 45 instances, for which all the solvers find a solution, into three different sets namely 'small spread', 'medium spread', and 'large spread'. The percentage of gaps shown in Table B.19 of Appendix B are used to group the instances.

We use two measures to evaluate the performance of different solvers.

Our first measure is the number of instances for which a solver finds a better solution than the solutions obtained by other solvers.

Our second measure is the percentage of gap as defined in § 4.2.1. We calculate the average and the standard deviation of percentage of gaps over a group of instances. Here, we do not use the percentage of improvement as a measure since the solvers do not start with the same initial solution.

Table 5.1- 5.3 summarize the performance of NPGC, with respect to the first measure, against Cplex-D, LB, and RINS. Table 5.4 shows the average and the standard deviation of percentage of gaps obtained by different solvers.

Table 5.1: Cplex-D versus NPGC on 45 benchmark instances for which both find solutions. Cplex-D better: the number of instances at which Cplex-D finds better solution than NPGC. NPGC better: the number of instances at which NPGC finds better solution than Cplex-D. Tied: the number of instances at which both Cplex-D and NPGC find the same improved solution. Entries indicate number of instances.

Cplex-D better	NPGC better	Tied
17	23	5

Table 5.2: LB versus NPGC on 45 benchmark instances for which both find solutions. Entries indicate number of instances.

LB better	NPGC better	Tied
18	21	6

Table 5.3: RINS versus NPGC on 45 benchmark instances for which both find solutions. Entries indicate number of instances.

RINS better	NPGC better	Tied
28	11	6

Table 5.4: The average ($\overline{x}$) and the standard deviation (δ) of percentage of gaps obtained by Cplex-D, LB, RINS, and NPGC on 45 benchmark instances for which each solver finds a solution.

	Cplex-D		LB		RINS		NPGC	
	$\overline{x}$	δ	$\overline{x}$	δ	$\overline{x}$	δ	$\overline{x}$	δ
On all 45 instances	63.45	235.51	73.37	231.73	71.61	315.00	153.82	712.71
On 32 small spread instances	1.50	2.01	1.41	1.99	0.77	1.32	1.29	1.86
On 7 medium spread instances	20.02	19.73	20.97	22.73	12.51	19.25	24.31	22.24
On 6 large spread instances	444.53	534.92	518.35	444.17	518.40	772.10	1117.82	1783.76

Analyzing the results shown on Table 5.1- 5.4, we find that on the chosen benchmark instances, NPGC is slightly better than Cplex-D and LB in small spread instances, is competitive with Cplex-

D and LB in medium spread instances, but worse than Cplex-D and LB in large spread instances.

Unfortuantely, NPGC is worse than RINS on these benchmark instances.

Chapter 6

Generating Hard Integer Program Instances

While there exists an established set of benchmark instances on which different heuristics can be compared, for example as mentioned in § 2.3.1, it is of interest to find new hard instances that might be added to the benchmark suite. In particular, due to the recent development of new feasibility heuristics, it is of interest to find new feasibility-hard instances. In 1998, Cornuéjols and Dawande showed how to pseudo-randomly generate both optimality-hard and feasibility-hard instances from different variations of Williams's market-sharing problem. In this chapter, we show how to pseudo-randomly generate a related class of instances that are both feasibility-hard and optimality-hard.

Before presenting our new class of instances, we review the Cornuéjols-Dawande feasibility-hard instances.

6.1 Cornuéjols-Dawande Feasibility-Hard Instances

Recall from § 2.3.2 that, for instances with n variables and m constraints, Cornuéjols-Dawande chose $n = 10(m-1)$ for generating their optimality-hard instances. Also recall from § 2.3.2 that dropping the slack/surplus variables from the constraints, every Cornuéjols-Dawande optimality-

hard instance gives rise to a Cornuéjols-Dawande feasibility-hard instance.

Using probability measures, namely the probability that a generated instance is infeasible and the expected number of solutions of a generated instance, Aardal et al. [1] showed that the Cornuéjols-Dawande feasibility-hard instances generated with $n = 10(m-1)$ are with high probability infeasible.

Since our objective is to generate instances for the purpose of evaluating heuristics, the generated instances should have at least one feasible solution, and finding any such solution should be hard. Therefore, the relation $n = 10(m-1)$ is not suitable for us.

Following the analysis of Aardal et al. Table 6.1 shows the probability measures for some n and m. We present a short form of this table in § 3.4.5.

Table 6.1: Probability measures for the Cornuéjols-Dawande feasibility-hard instances generated with different n and m. The values are obtained using the analysis of Aardal et al. [1].

Problem size		Probability of being infeasible	Expected number of solutions
n	m		
10	1	0.01	4.44
10	2	0.97	0.03
10	3	0.99	2.1e-4
20	1	0.00	3.24e+3
20	2	2.8e-7	15.05
20	3	0.93	0.08
20	4	0.99	4.1e-4
30	1	0.00	2.7e+6

Continued on next page

Table 6.1 – Continued from previous page

Problem size		Probability of	Expected number
n	m	being infeasible	of solutions
30	2	0.00	1.0e+4
30	3	1.4e-19	43.41
30	4	0.83	0.19
30	5	0.99	8.6e-4
40	1	0.00	2.4e+9
40	2	0.00	7.9e+6
40	3	0.00	2.8e+4
40	4	8.4e-49	110.69
40	5	0.647	0.434
40	6	0.998	1.73e-3

In order to generate feasibility-hard instances with at least one feasible solution, it appears from this table that a good choice for the relation between n and m is to pick $n = 10m$, since for this choice, the probability of a generated instance being infeasible is close to zero, and the expected number of solutions is small as well as larger than one. In § 3.4.5 we presented experimental results comparing feasibility heuristics on some pseudo-randomly generated Cornuéjols-Dawande feasibility-hard instances using $n = 10m$. However, we found in experiments that with this choice of $n = 10m$, many pseudo-randomly generated instances were infeasible. Assume that one wants to generate feasibility-hard instances with 30 variables. Then following the probability measures, there are only three options for choosing m. It can be either 3 or 2 or 1, and we see that the expected number of solutions increases quite rapidly as m changes from 3 to 2 to 1. This rapid increase in expected number of solutions may rapidly reduce the difficulty of solving instances.

This observation led us to introduce a constrained version of the Williams's market-sharing problem for which we have more options in choosing m for a particular n, and for which the expected number of solutions changes much more smoothly as m changes. In the next section we introduce this constrained problem and a way to pseudo-randomly generate instances from the problem. Then we will show that the new class of instances has the aforementioned properties.

6.2 Constrained Williams's Market-Sharing Problems

We use the following constrained version of Williams's market-sharing problem to pseudo-randomly generate our new class of instances. The difference between this problem and the original version of Williams's market-sharing problem are indicated by italics.

> A large company has two divisions D_1 and D_2. The company supplies retailers with several products. The goal is to allocate each retailer to either division D_1 or division D_2 so that D_1 controls y% of the company's market for each product and D_2 the remaining (100-y)% or, if such a perfect $y/(100-y)$ split is not possible for all the products, the goal is to minimize the sum of percentage deviations from the desired split *with the new imposed constraints that D_1 has a specific choice to control less than y% of company's market for some (say m_1) products and a specific choice to control greater than y% of company's market for remaining $(m - m_1)$ products.*

We model this problem as the following integer program,

$$
\begin{array}{lll}
\min & \sum_{i=1}^{m} s_i & \\
\text{s.t.} & \sum_{j=1}^{n} a_{ij}x_j + s_i = b_i & i = 1, ..., m_1 \\
& \sum_{j=1}^{n} a_{ij}x_j - s_i = b_i & i = (m_1 + 1), ..., m \\
& x_j \in \{0, 1\} & j = 1, ..., n \\
& s_i \geq 0 & i = 1, ..., m
\end{array}
$$

where n, m, a_{ij} are the number of retailers, the number of products, and the demand of retailer j for product i respectively. For the desired $y/(100-y)$ split, $b_i = \lfloor f \sum_{j=1}^{n} a_{ij} \rfloor$, where $f = \frac{y}{100}$.

With respect to the desired split, the first m_1 constraints specify the condition on the products for which D_1 has a choice to under-produce, and the remaining constraints specify the condition on the products for which D_1 has a choice to over-produce. The motivation of our construction comes from the observation that most of the 2^n possibilities for vector x which satisfies the first m_1 constraints violates the remaining constraints and vice versa; therefore, the instances of this construction are feasible only for a few choice over x. Since these inequality constraints are less stringent than the equality constraints of Cornuéjols-Dawande feasibility-hard instances, our intuition was that there might be more options for choosing m for a particular n. As a result, the expected number of solutions of a generated instance might change smoothly as m changes. We will show later that the computed probability measures support our intuition.

As with the Cornuéjols-Dawande feasibility-hard instances, for constrained market-sharing feasibility-hard instances, we choose a split of 50/50 and each integer a_{ij} uniformly between 0 and $(d-1)$, where $d = 100$. A 50/50 split makes f equal to 1/2. We set $m_1 = \lceil pm \rceil$ when $0 < p \leq 0.5$ or $m_1 = \lfloor pm \rfloor$ when $0.5 < p < 1$. Notice that, for some p', using $p = p'$ or $p = 1 - p'$ generates similar hard instances, since the value of m_1 with $p = p'$ becomes the value of $(m - m_1)$ with $p = 1 - p'$ and vice versa. We will show later what value should be chosen for p.

The complexity of finding a feasible solution for an instance of the proposed problem class is equivalent to the complexity of answering the yes/no question "Is there a solution satisfying the constraints mentioned in the above formulation?"; we can show this to be NP-complete when $m = 2$. When $m = 2$, for any p with $0 < p < 1$, an instance of the proposed problem class contains the two constraints $\sum_{j=1}^{n} a_{1j}x_j + s_1 = b_1$ and $\sum_{j=1}^{n} a_{2j}x_j - s_2 = b_2$. In the case where $a_{1j} = a_{2j}$ for all j, the instance is equivalent to finding a solution to $\sum_{j=1}^{n} a_{1j}x_j = b_1$ which is the subset sum problem, which is NP-complete [36].

Recall that our objective is to generate instances that have feasible solutions with high probability but for which finding any such solutions is hard. Therefore, following the probability analysis of Aardal et.al. [1], for a generated instance of our proposed constrained market-sharing problem we compute the probability of being infeasible and the expected number of solutions. Our anal-

ysis, which is similar to the analysis of Aardal et al. for the Cornuéjols-Dawande feasibility-hard instances, is described in the next sections.

6.2.1 The Expected Number of Solutions

As in [1], we consider that a_{ij} are uniformly distributed integers from the set $\{0, \cdots, d-1\}$, and $b_i = \lfloor f \sum_{j=1}^{n} a_{ij} \rfloor$.

Consider each possible solution x of an instance as a subset $S \subseteq \{1, 2, \cdots, n\}$ where $x_j = 1$ if $j \in S$, and $x_j = 0$ otherwise.

In order to determine the probability that a vector x satisfies row i of a generated instance, consider a random variable $z_i(S) = \sum_{j \in S} a_{ij} - \lfloor f \sum_{j=1}^{n} a_{ij} \rfloor$ representing the difference between the values of the both sides of row i, which is equivalent to the absolute value of the slack/surplus variable s_i. The probability that x satisfies a row of type $\sum_{j=1}^{n} a_{ij} x_j + s_i = b_i$ is denoted by $Pr[z_i(S) \leq 0]$, and that x satisfies a row of type $\sum_{j=1}^{n} a_{ij} x_j - s_i = b_i$ is denoted by $Pr[z_i(S) \geq 0]$.

Following Aardal et al. assume that the random variable $y_i(S) = \sum_{j \in S} a_{ij} - f \sum_{j=1}^{n} a_{ij}$, and the random variable $u_i = f \sum_{j=1}^{n} a_{ij} - \lfloor f \sum_{j=1}^{n} a_{ij} \rfloor$. Thus, $z_i(S) = y_i(S) + u_i$. For any rational number $f = g/h$, where g and h are relatively prime positive natural numbers, $y_i(S) = \frac{k}{h}$ for some integer k, and $u_i = \frac{k'}{h}$ for some integer k' between 0 and $(h-1)$. Therefore, we have

$$Pr[z_i(S) \leq 0] = Pr[y_i(S) \leq -u_i] = \sum_{k=0}^{\infty} Pr\left[y_i(S) = -\frac{k}{h}\right],$$

and

$$Pr[z_i(S) \geq 0] = Pr[y_i(S) \geq -u_i] = \sum_{k=-\infty}^{h-1} Pr\left[y_i(S) = -\frac{k}{h}\right].$$

Aardal et al. derived the probability generating function of $y_i(S)$ as the following.

$$G_{y_i(S)}(x) = \frac{1}{d^n} \frac{1}{x^{f(d-1)(n-|S|)}} \left(\frac{x^{(1-f)d} - 1}{x^{(1-f)} - 1}\right)^{|S|} \left(\frac{x^{fd} - 1}{x^f - 1}\right)^{n-|S|},$$

which has the expansion of the form $\sum_j c_j x^{j/h}$, where $c_j = Pr[y_i(S) = \frac{j}{h}]$. For $f = \frac{1}{2}$, the probability generating function becomes the following.

$$G_{y_i(S)}(x) = \frac{1}{d^n} \frac{1}{x^{(d-1)(n-|S|)/2}} \left(\frac{x^{d/2}-1}{x^{1/2}-1} \right)^n .$$

We use the following lemma to find the Taylor expansion of the factors in the above expression.

Lemma 6.2.1 *(Aardal, Bixby, Hurkens, Lenstra, and Smeltink [1])*

$((y^d-1)/(y-1))^n = \sum_{j=0}^{\infty} a_j y^j$, *where*

$$a_j = \sum_{k=0}^{min\{n,\lfloor j/d \rfloor\}} \binom{n}{k} (-1)^k \binom{j-dk+n-1}{j-dk}.$$

Also, notice that $a_j = 0$ *for* $j > (d-1)n$.

Since c_j depends on d, n, and the size of S, we denote $Pr[z_i(S) \leq 0]$ by $q_1(n, d, |S|)$, where

$$q_1(n, d, |S|) = \frac{1}{d^n} \sum_{k=0}^{(d-1)(n-|S|)} a_k,$$

and $Pr[z_i(S) \geq 0]$ by $q_2(n, d, |S|)$, where

$$q_2(n, d, |S|) = \frac{1}{d^n} \sum_{k=(d-1)(n-|S|)-1}^{(d-1)n} a_k.$$

Now, the probability that a vector x constitutes a feasible solution for a generated instance is $q_1(n, d, |S|)^{m_1} q_2(n, d, |S|)^{m-m_1}$. Therefore, the expected number of solutions,

$$\begin{aligned} E[\text{number of solutions}] &= \sum_{S \subseteq \{1,\cdots,n\}} q_1(n, d, |S|)^{m_1} q_2(n, d, |S|)^{m-m_1} \\ &= \sum_{s=0}^{n} \alpha q_1(n, d, s)^{m_1} q_2(n, d, s)^{m-m_1}, \end{aligned}$$

where $s = |S|$ and $\alpha = \binom{n}{s}$.

6.2.2 Probability of Generating Infeasible Instances

For simplicity we assume that each distinct subset S is independent. Thus the probability of generating infeasible instances becomes,

$$\begin{aligned} Pr[\text{number of solution is zero}] &= Pr[S \text{ yields no solution}, \forall S \subseteq \{1, 2, \cdots, n\}] \\ &\approx \prod_{S \subseteq \{1,\cdots,n\}} Pr[S \text{ yields no solution}] \\ &= \prod_{S \subseteq \{1,\cdots,n\}} (1 - q_1(n, d, |S|)^{m_1} q_2(n, d, |S|)^{m-m_1}) \\ &= \prod_{s=0}^{n} ((1 - q_1(n, d, s)^{m_1} q_2(n, d, s)^{m-m_1}))^{\alpha}, \end{aligned}$$

where $s = |S|$ and $\alpha = \binom{n}{s}$.

Following the above computation, Table 6.2 shows the probability measures for constrained market-sharing instances. Since we want to show these measures as the relation of $\frac{n}{m}$ changes, we introduce a new variable k and define m to be $\lfloor \frac{n}{k} \rfloor$.

Table 6.2: Probability measures for the constrained market-sharing instances. PI: the probability of a generated instance being infeasible, ES: the expected number of solutions, n, p: the parameters described in the problem formulation, k: a variable used to define m.

$p = 0.9$						
	$n = 10$		$n = 20$		$n = 30$	
k	PI	ES	PI	ES	PI	ES
0.55	0.540	0.613	0.352	1.042	0.348	3.356
0.6	0.297	1.209	0.054	2.918	3.4e-8	17.18
0.65	0.181	1.701	2.7e-4	8.196	1.1e-20	45.90
0.7	0.090	2.396	7.8e-8	16.35	5.8e-75	170.9
0.75	0.033	3.383	6.2e-15	32.69	2.7e-144	330.54
0.8	0.008	4.787	3.4e-29	65.53	0.000	891.8

Continued on next page

Table 6.2 – Continued from previous page

	$n = 10$		$n = 20$		$n = 30$	
k	PI	ES	PI	ES	PI	ES
0.85	0.001	6.794	4.6e-41	92.86	0.000	1.7e+3
0.9	0.001	6.794	6.3e-58	131.6	0.000	3.3e+3
0.95	5.3e-5	9.681	6.6e-82	186.8	0.000	6.6e+3
1.0	5.3e-5	9.681	5.1e-116	265.3	0.000	9.2e+3
$p = 0.5$						
1.1	0.567	0.566	0.431	0.839	0.243	1.414
1.2	0.314	1.152	0.032	3.412	1.0e-5	11.46
1.3	0.092	2.371	0.001	6.903	9.5e-11	23.07
1.4	0.092	2.371	8.2e-7	14.01	1.6e-41	93.92
1.5	0.007	4.943	4.0e-13	28.54	3.1e-83	189.9
1.6	0.007	4.943	4.3e-26	58.38	4.5e-340	781.3
1.7	2.4e-5	10.49	7.3e-53	120.0	0.000	1.5e+3
1.8	2.4e-5	10.49	7.3e-53	120.0	0.000	3.2e+3
1.9	2.4e-5	10.49	1.7e-108	248.0	0.000	6.6e+3
2.0	2.4e-5	10.49	1.7e-108	248.0	0.000	6.6e+3

The values shown in Table 6.2 suggest that, for a fixed value of p, since the expected number of solutions increases as k increases, the hardness of finding a feasible solution should decrease. Experiments with various solvers support this claim.

The information in Table 6.2 also suggests that the smallest hard instances, in terms of nm, of this class occurs when $p = 0.5$, since for a fixed n, almost similar expected number of solutions are obtained at larger value of k for $p = 0.5$ than for $p > 0.5$. For example, the expected number of solutions of the instances generated with $p = 0.9, n = 30, k = 0.7$ is close to the expected number

of solutions of the instances generated with for $p = 0.5, n = 30, k = 1.5$. However, the size of instances in terms of nm is $30 \times \lfloor \frac{30}{0.7} \rfloor = 1260$ for the setting with $p = 0.9$, whereas the size of instances in terms of nm is $30 \times \lfloor \frac{30}{1.5} \rfloor = 600$ for the setting with $p = 0.5$.

Comparing the probability measures for Cornuéjols-Dawande feasibility-hard instances and our constrained market sharing instances, notice that for a fixed n, the expected number of solutions changes more smoothly in the constrained market sharing instances as m changes. For example, considering $p = 0.5$, to generate constrained market-sharing instances with 30 variables, m can be any number from 1 to $\lfloor \frac{30}{1.1} \rfloor = 27$. By contrast, m can be only between 1 and 3 for Cornuéjols-Dawande feasibility-hard instances.

In the next section, we present the performance of different solvers on some pseudo-randomly generated constrained market-sharing instances.

6.3 Solver Performance on Constrained Market-Sharing Instances

Based on the probability measures shown in Table 6.2, we chose $k = 2, k = 1.5$, and $k = 1.3$ with $p = 0.5$ to generate instances. We did not decrease k further since even with $k = 1.3$, experiments showed that many instances generated were infeasible.

For generating pseudo-random numbers a_{ij} with uniform distribution, we used the 'rand()' function from the C programming language. The rand() function gives an integer in the $[0, M]$, where M is compiler dependent. For example, M was 2147483647 in the compiler we used. The rand() function uses the linear congruential generator algorithm [32], one of the best known pseudo-random number generators. The expression $\frac{rand()}{M+1}$ gives a pseudo-random floating point number in the interval [0, 1). For generating integers in the interval [0, 99], we used the integer part of $100 * \frac{rand()}{M+1}$.

Setting $k = 1.3$ and $p = 0.5$, we generated 100 pseudo-random instances with 50, 75, and 100 variables each; setting $k = 1.5$ and $p = 0.5$, we generated 100 pseudo-random instances with 50, 75, and 100 variables each; setting $k = 2$ and $p = 0.5$, we generated 100 pseudo-random instances

with 50, 100, and 150 variables each.

On these generated instances, we applied each of the five solvers, namely Cplex-D, Cplex-F, PBS4, FP, and PGC, with one CPU-hour time limit to find an initial feasible solution. We ran all the experiments on an 2403 MHz AMD Athlon processor with 128 MByte of memory under Redhat Linux 9.0.

Table 6.3 summarizes the performance of FP, Cplex-D, Cplex-F, PBS4, and PGC_1.

Table 6.3: The performance summary of different solvers to find a feasible solution on pseudo-randomly generated constrained market-sharing instances. The measure 'successful' indicates the number of instances for which a solver finds a feasible solution. The measure 'takes least amount of time in' indicates the number of instances for which a solver takes the least amount of time among the five solvers.

Problem size				FP	Cplex-D	Cplex-F	PBS4	PGC_1
n	k	m						
50	2.0	25	successful	100	0	100	64	100
			among the 100 instances at which at least one solver succeeds					
			takes least amount of time in	1	0	0	0	99
100	2.0	50	successful	96	0	95	0	100
			among the 100 instances at which at least one solver succeeds					
			takes least amount of time in	1	0	0	0	99
150	2.0	75	successful	55	0	0	0	100
			among the 100 instances at which at least one solver succeeds					
			takes least amount of time in	0	0	0	0	100

Continued on next page

Table 6.3 – Continued from previous page

Problem size				FP	Cplex-D	Cplex-F	PBS4	PGC_1
n	k	m						
50	1.5	33	successful	94	0	97	1	97
			among the 97 instances at which at least one solver succeeds					
			takes least amount of time in	2	0	10	0	88
75	1.5	50	successful	54	0	73	0	90
			among the 90 instances at which at least one solver succeeds					
			takes least amount of time in	2	0	3	0	85
100	1.5	66	successful	15	0	10	0	83
			among the 83 instances at which at least one solver succeeds					
			takes least amount of time in	0	0	0	0	83
50	1.3	38	successful	63	0	67	0	67
			among the 67 instances at which at least one solver succeeds					
			takes least amount of time in	1	0	14	0	52
75	1.3	57	successful	17	0	32	0	50
			among the 50 instances at which at least one solver succeeds					
			takes least amount of time in	2	0	2	0	48
100	1.3	76	successful	1	0	0	0	31
			among the 31 instances at which at least one solver succeeds					
			takes least amount of time in	0	0	0	0	31

In the alloted one CPU-hour, Cplex-D found 3 and 33 infeasible instances among the 100 instances generated with $n = 50, k = 1.5$ and $n = 50, k = 1.3$ respectively. Therefore, from the

experiments, it seems that some instances generated with $k = 1.5$ and $k = 1.3$ may have no feasible solution. For this reason, Table 6.3 shows the comparison only on the number of instances for which at least one solver finds a feasible solution within the alloted time. The results shown on this table suggest that, it is difficult for the solvers to find a solution for the constrained market-sharing instances as the number of variable goes beyond roughly 150 when $k = 2.0$, 100 when $k = 1.5$, and 75 when $k = 1.3$.

Notice that, we can also use the constrained market-sharing instances as optimality-hard instances without changing the model, whereas we need to change the model to obtain optimality-hard instances from Cornuéjols-Dawande feasibility-hard instances. In § 4.2.5, we have seen that it is also difficult to find an optimal solution for the constrained market-sharing instances.

Our new feasibility heuristic PGC shows much stronger performance than other solvers on constrained market-sharing instances, but shows much worse performance than Cplex on Cornuéjols-Dawande feasibility-hard instances. We do not know the reason for this difference in PGC's performance. However, we point out the following differences between these two class of instances which might have a role in PGC's performance.

One difference is that the feasible region corresponding to the linear programming relaxation of a Cornuéjols-Dawande feasibility-hard instance is a hyperplane generated by the intersection of several hyperplanes, whereas that of a constrained market-sharing instance is a region bounded by several hyperplanes. PGC might handle those instances better for which the feasible region is bounded by hyperplanes rather than intersection of hyperplanes.

Another difference is that the Cornuéjols-Dawande feasibility-hard instances have no slack/surplus variables in the initial formulation, whereas constrained market-sharing instances have many slack/surplus variables in the initial formulation. PGC introduces new surplus variables only when they add Gomory cuts. Therefore, PGC, at the beginning, fails to find Type 1 pivots in the Cornuéjols-Dawande feasibility-hard instances; it performs only Type 2 pivots which does not bring integer variables out of the basis. In contrast, PGC, at the beginning, may perform Type 1 pivots in the constrained market-sharing instances.

Chapter 7

Conclusions

It is of interest to develop heuristics that will be effective for any given mixed integer program irrespective of its underlying form. If we consider a particular optimization problem such as the set cover problem, then all the inequalities have the same form, that is $\sum_{j=1}^{n} a_{ij}x_j \geq 1$; the set packing problem, then all the inequalities have the form $\sum_{j=1}^{n} a_{ij}x_j \leq 1$; the set partition problem, then all the inequalities have the form $\sum_{j=1}^{n} a_{ij}x_j = 1$. However, most real world problems come with inequalities having different forms. In this thesis, we have presented three heuristics for general mixed integer programs.

We have introduced a new feasibility heuristic, PGC. It uses the simplex tableau pivoting framework of PC. PGC replaces the PC restart phase, which fails in a large number of instances, with a Gomory cut based restart phase, which is guaranteed to succeed. PGC also replaces the PC pivot rules, focused on finding solutions with better objective value, with a set of Gomory cut based pivot rules, focused on finding feasible solutions as quickly as possible. We have chosen Gomory cuts as the cutting planes to use in PGC since they are easy to derive from a simplex tableau. Experimental results suggest that PGC is a competitive alternative to the recent heuristic FP. Besides this, on a set of pseudo-randomly generated hard 0-1 instances, PGC outperforms all the considered solvers, namely PBS4, FP, and the commercial solver Cplex. We were not able to show that PGC terminates in a finite number of steps. However, we hope that, like the recent heuristic FP which has no finite

convergence property but practically works well, the introduced PGC has the merit to stand as a competitive heuristic in practice. The introduction of PGC raises several open questions that would be interesting to answer. Is the number of Gomory cuts added in PGC finite? Would generating Gomory cuts from the rows corresponding to other integer-infeasible integer-constrained variables, than the row corresponding to the most integer-infeasible integer-constrained variable, give a better variant of PGC? Would incorporating some other cuts such as lift-and-project cuts, those are not obtained in a straight-forward way from the simplex tableau, in place of Gomory cuts, result a better heuristic? Would incorporating PGC in different nodes of branch-and-bound or branch-and-cut like exact solvers be beneficial?

We have also introduced a new improvement heuristic, DINS. It defines neighbourhoods, based on a distance metric, using ideas from the LB and RINS as well as some new ideas. These ideas include changing the bounds of the integer-constrained variables and using the history of integer-constrained variables in the mixed integer program search tree. Experimental results suggest that DINS performs better than existing comparable heuristics. Experimental results also suggest that exploring useful small neighbourhoods as in DINS is better in the long run than exploring useful large neighbourhoods as in RINS, but worse at the initial level of computation. Future works could include establishing a hybrid strategy of small neighbourhood search and large neighbourhood search that might come out as a better option than using only one type of neighbourhood search. It would be also interesting to know if there is any better way to model the distance metric used in DINS.

We have also introduced NPGC, a 'find-and-improve' heuristic. NPGC is extention of PGC that is similar in approach to the implementation of LB incorporated in a mixed integer program search tree. In such an implementation, LB is applied whenever the mixed integer program search tree, which is generated by branch-and-bound or branch-and-cut like exact solver, finds a new feasible solution. NPGC replaces the mixed integer program search tree by its Gomory cut based neighbourhood search. In other words, in NPGC, LB is applied whenever the Gomory cut based neighbourhood search finds a new feasible solution. Though NPGC is not better than the compara-

ble heuristic RINS, it is a natural extension of PGC that finds feasible solutions for some instances in a period of time in which the commercial solver Cplex fails to find such solutions.

We have also presented a new class of hard 0-1 instances, namely constrained market-sharing instances, obtained by modifying Williams's Market-sharing problem and using the idea of Cornuéjols-Dawande optimality-hard instances. PGC appears significantly stronger than any other heuristics when tested on these constrained market-sharing instances. It would be interesting to see a mathematical reasoning behind this performance, which might help us to categorize the class of instances for which PGC works well.

As a final remark, we hope that this thesis has increased the state-of-the-art for mixed integer program heuristics by introducing three new heuristics, and showing several new directions for further research.

Bibliography

[1] K. Aardal, R.E. Bixby, C.A.J. Hurkens, A.K. Lenstra, and J.W. Smeltink. Market split and basis reduction: Towards a solution of the Cornuéjols-Dawande instances. *INFORMS Journal on Computing*, 12(3):192–202, 2000.

[2] F. Aloul and B. Al-Rawi. Pseudo-boolean solver version 4. 2005. http://www.eecs.umich.edu/ faloul/Tools/pbs4.

[3] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack search pseudo-boolean solver. *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2002.

[4] F.A. Aloul, A. Ramani, I.L. Markov, and K.A. Sakallah. Generic ILP versus Specialized 0-1 ILP: An update. *International Conference on Computer Aided Design*, pages 450–457, 2002.

[5] K. Anderson, G. Cornuéjols, and Y. Li. Reduce-and-split cuts: Improving the performance of mixed integer Gomory cuts. *Management Science*, 51:1720–1732, 2005.

[6] D. Avis and V. Chvátal. Notes on Bland's pivoting rule. *Mathematical Programming Study*, 8:24–34, 1978.

[7] E. Balas. Intersection cuts – a new type of cutting planes for integer programming. *Operations Research*, 19:19–39, 1971.

[8] E. Balas. Disjunctive programming. *Annals of Discrete Mathematics*, 5:3–51, 1979.

[9] E. Balas, S. Ceria, and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58:295–324, 1993.

[10] E. Balas, S. Ceria, and G. Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42(9):1229–1246, 1996.

[11] E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19:1–9, 1996.

[12] E. Balas, S. Ceria, M. Dawande, F. Margot, and G. Pataki. Octane: a new heuristic for pure 0-1 programs. *Operations Research*, 49(2):207–225, 2001.

[13] E. Balas and C.H. Martin. Pivot and complement – a heuristic for 0-1 programming. *Management Science*, 26(1):86–96, 1980.

[14] E. Balas and C.H. Martin. Pivot and shift – a heuristic for mixed integer programming. Technical report, GSIA, Carnegie Mellon University, 1986.

[15] E. Balas and M. Perregaard. A precise correspondence between lift-and-project cuts, simple disjunctive cuts and mixed integer Gomory cuts for 0-1 programming. *Mathematical Programming B*, 94:221–245, 2003.

[16] E. Balas, S. Schmieta, and C. Wallace. Pivot and shift – a mixed integer programming heuristic. *Discrete Optimization*, 1:3–12, 2004.

[17] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-boolean optimization. Technical report, Max-Plank-Institut Für Informatik, 1995.

[18] R.E. Bixby, E.A. Boyd, and R.R. Indovina. MIPLIB: A test set of mixed integer programming problems. *SIAM News*, 25(20):16, March 1992.

[19] R.E. Bixby, S. Ceria, C.M. McZeal, and M.W.P. Savelsberg. An updated mixed integer programming library. MIPLIB 3.0, 1998. Department of Computational and Applied Mathematics, Rice University, Web address: http://www.caam.rice.edu/ bixby/miplib/miplib.html.

[20] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. *Procedure of the 40th Design Automation Conference*, pages 830–835, 2003.

[21] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4:305–337, 1973.

[22] V. Chvátal. Hard knapsack problems. *Operations Research*, 28:1402–1411, 1980.

[23] V. Chvátal. *Linear Programming*. W.H. Freeman and Company, New York, 1983.

[24] W. Cook, T. Rutherford, H.E. Scarf, and D. Shallcross. An implementation of the generalized basis reduction algorithm for integer programming. *ORSA Journal on Computing*, 5(2):206–212, 1993.

[25] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. John Wiley & Sons, 1998.

[26] G. Cornuéjols. Revival of the gomory cuts in the 1990's. to be appeared on State of the Art and Recenet Advances in Integer Programming.

[27] G. Cornuéjols and M. Dawande. A class of hard small 0-1 programs. *6th IPCO, Lecture Notes in Computer Science*, 1412:284–293, 1998.

[28] H. Crowder, E.L. Johnson, and M. Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31(5):803–834, 1983.

[29] E. Danna, E. Rothberg, and C.L. Pape. Exploring relaxation induced neighborhhods to improve mip solutions. *Mathematical Programming*, 102:71–90, 2005.

[30] G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In Tj. C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. Wiley, New York, 1951.

[31] DEIS. Library of instances. http://www.or.deis.unibo.it/research_pages/ORinstances/ MIPs.html.

[32] D.E.Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1997.

[33] B.H. Faaland and F.S. Hillier. Interior path methods for heuristic integer programming procedures. *Operations Research*, 27(6):1069–1087, 1979.

[34] M. Fischetti, F. Glover, and A. Lodi. The feasibilty pump. to be appeared on Mathematical Programming.

[35] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming B*, 98:23–49, 2003.

[36] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[37] S. Ghosh. Distance induced neighbourhood search - a mixed integer programming heuristic. 2006. In preparation.

[38] S. Ghosh and R. Hayward. Pivot and Gomory cut: a mixed integer programming heuristic. 2006. In preparation.

[39] D. Goldfarb and W.Y. Sit. Worst case behavior of the steepest edge simplex method. 1:277–285, 1979.

[40] R.E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the AMS*, 64:275–278, 1958.

[41] R.E. Gomory. An algorithm for the mixed integer problem. Technical Report RM-2597, The Rand Corporation, Santa Monica, CA, 1960.

[42] U.-U. Haus, M. Köppe, and R. Weismantel. The integral basis method for integer programming. *Mathematical Methods of Operations Research*, 53:353–361, 2001.

[43] F.S. Hillier. Efficient heuristic procedures for integer linear programming with an interior. *Operations Research*, 17(4):600–637, 1969.

[44] K.L. Hoffman and M. Padberg. Solving airline crew scheduling problems by branch-and cut. *Management Science*, 39(6):657–682, 1993.

[45] T. Ibaraki, T. Ohashi, and H. Mine. A heuristic algorithm for mixed integer programming problems. *Mathematical Programming Study*, 2:115–136, 1974.

[46] R.G. Jerslow. The simplex algorithm with the pivot rule of maximizing criterion improvement. 4:367–377, 1973.

[47] N. Karmarker. A new polynomial-time algorithm for linear programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing (Washington, 1984)*, pages 302–311. The Association of Computing Machinery, New York, 1984. [also: *Combinatorica* 4 (1984) pp.373-395].

[48] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[49] L.G. Khachiyan. A polynomial algorithm in linear programming (in russian). *Doklady Academi Nauk SSSR*, 244:1093–1096, 1979. [English translation: Soviet Mathematics Doklady 20 (1979) pp. 191-194].

[50] L.G. Khachiyan. Polynomial algorithms in linear programming (in russian). *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 20:51–68, 1980. [English translation: U.S.S.R. Computational Mathematics and Mathematical physics 20 (1980) pp. 53-72].

[51] V. Klee and G.J. Minty. How good is the simplex algorithm? In O. Shisha, editor, *Inequalities III*, pages 159–175. Academic Press, New York, 1972.

[52] A.H. Land and A.G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[53] L.Lovász and H.E. Scarf. The generalized basis reduction algorithm. *Mathematics of Operations Research*, 17:751–764, 1992.

[54] A. Løkketangen and F. Glover. Solving zero/one mixed integer programming problems using tabu search. *European Journal of Operational Research*, 106:624–658, 1998.

[55] A. Løkketangen, K. Jörnsten, and S. Storøy. Tabu search within a pivot and complement framework. *International Transactions in Operational Research*, 1(3):305–317, 1994.

[56] Y.S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient SAT solver. *In selected papers from SAT 2004, Lecture notes in Computer Science*, 3542:360–375, 2005.

[57] J.P. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transaction on Computers*, 48:506–521, 1999.

[58] A. Martin, T. Achterberg, and T. Koch. Miplib 2003. http://miplib.zib.de.

[59] S. Mehrotra. On the implementation of a Primal-Dual interior point method. *SIAM Jornal of Optimization*, 2:575–601, 1992.

[60] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *Procedure of Design Automation Conference*, 2001.

[61] M. Nediak and J. Eckstein. Pivot, cut, and dive: A heuristic for mixed 0-1 integer programming. *RUTCOR Research Report*, RRR 53-2001, 2001.

[62] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[63] G.L. Nemhauser and L.A. Wolsey. Integer programming. In G.L. Nemhauser, A.H.G. Rinnooy, and M.J. Todd, editors, *Handbooks in OPerations Research and Management Science 1: Optimization*, pages 447–527. North-Holland, Amsterdam, 1989.

[64] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33:60–100, 1991.

[65] R.G. Parker and R.L. Rardin. *Discrete Optimization*. Academic Press, New York, 1988.

[66] S.D. Prestwich. Randomised backtracking for linear pseudo-boolean constraint problems. *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems*, pages 7–20, 2002.

[67] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[68] V.V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin, 2001.

[69] J.P. Walser. Solving linear pseudo-boolean constraints with local search. *Proceedings of the Eleventh Conference on Artificial Intelligence*, pages 269–274, 1997.

[70] X. Wang. *A new implementation of the generalized basis reduction algorithm for convex integer programming*. PhD Thesis, Yale University, 1997.

[71] J.P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68:63–69, 1998.

[72] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. *Procedure of the Design Automation Conference*, 2001.

[73] H.P. Williams. *Model Building in Mathematical Programming*. Wiley, 1978.

[74] L.A. Wolsey. *Integer Programming*. John Wiley & Sons, 1998.

[75] H. Zhang. SATO: An efficient propositional prover. *International Conference on Automated Deduction*, 1997.

Index

Approximation Algorithms, 8

Benchmark Instances, 25
Branch-and-Bound, 9, 10
Branch-and-Cut, 11

Constrained Market-sharing Problem, 91
Convexity Cut, 20

Distance Induced Neighbourhood Search, 57

Feasibility Pump, 22

Gomory Cuts, 12

Integer Program, 4
    Complexity, 7
    Solution Methods, 2
        Exact, 2, 9
        Heuristic, 2, 15
Interior Path Method, 21

Linear Programming Relaxation, 5
Local Branching, 23

Market-sharing Problem, 26

Neighbourhood Pivot and Gomory Cut, 80

Octane, 17

Pivot and Complement, 15, 29
Pivot and Gomory Cut, 34
Pivot, Cut, and Dive, 18
Pivot-and-Shift, 16

Relaxation Induced Neighbourhood Search, 24

SAT Problem, 8
Simplex Method, 5
    Simplex Tableau, 6

Tabu Search, 20

Vertex Cut, 20

Appendix A

Pseudo-Code of PGC

Here we give a more detailed pseudo-code description of our heuristic algorithms PGC_0 and PGC_1. These two algorithms are composed of the following procedures.

Procedure Check_P_Feasible	Checks whether the current $\mathcal{L}(P^+)$ solution is P-feasible.
Procedure Round_Truncate	Checks whether rounding or truncating the current $\mathcal{L}(P^+)$ solution yields a P-feasible solution.
Procedure Generate_Gomory_Cut	Generates a cut from the tableau row corresponding to the most integer-infeasible integer-constrained variable.
Procedure Add_Gomory_Cut	Adds a Gomory cut to the simplex tableau.
Procedure PGC_0_Type1_Pivot	Searches for a PC Type 1 pivot and, if one is found, performs the pivot.
Procedure PGC_0_Type2_Pivot	Searches for a PC Type 2 pivot and, if one is found, performs the pivot.
Procedure PGC_1_Type1_Pivot	Searches for a PGC_1 Type 1 pivot and, if one is found, performs the pivot.
Procedure PGC_1_Type2_Pivot	Searches for a PGC_1 Type 2 pivot and, if one is found, performs the pivot.

Procedure Restart	Tries to cross the cut. If P has a feasible solution, it is guaranteed to be successful.
Procedure PGC$_0$	The main PGC$_0$ procedure.
Procedure PGC$_1$	The main PGC$_1$ procedure.

Procedure Check_P_Feasible
INPUT: the linear programming relaxation $\mathcal{L}(P^+)$ of P with the current basis, and the current basic feasible solution x^*.
OUTPUT: return *true* if x^* is P-feasible, else return *false*.

1. for each basic variables x_i do
2. if (the type of x_i restricts it to be integer and x_i^* is not integer) then
3. return *false*.
4. return *true*.

Procedure Round_Truncate
INPUT: the linear programming relaxation $\mathcal{L}(P^+)$ of P with the current basis, and the current basic feasible solution x^*.
OUTPUT: if successful, return a P-feasible solution x_{mip}, else return null.

1. $x_{mip} \leftarrow$ null.
2. store the information of current basis of $\mathcal{L}(P^+)$.
3. store the existing bound information of integer-constrained variables in $\mathcal{L}(P^+)$.
   CHECK ROUNDING
4. set new fixed bound for each of the integer constrained variables in $\mathcal{L}(P^+)$ to the nearest integer value of the respective value in x^*.
5. solve the $\mathcal{L}(P^+)$ using the bounded variable revised simplex algorithm.
6. if (a solution x_{mip} to $\mathcal{L}(P^+)$ is found) then
7. return x_{mip}.
   CHECK TRUNCATING
8. set new fixed bound for each of the integer constrained variables in $\mathcal{L}(P^+)$ to the largest integer value that is not larger than the respective value in x^*.
9. solve the $\mathcal{L}(P^+)$ using the bounded variable revised simplex algorithm.
10. if (a solution x_{mip} to $\mathcal{L}(P^+)$ is found) then
11. return x_{mip}.
12. restore the actual bound information of integer-constrained variables in $\mathcal{L}(P^+)$.
13. restore the current basis of $\mathcal{L}(P^+)$.
14. return x_{mip}.

Procedure Generate_Gomory_Cut

INPUT: the linear programming relaxation $\mathcal{L}(P^+)$ of P with the current basis, and the current basic feasible solution x^*.

OUTPUT: return a Gomory cut constructed from the row corresponding to the most integer-infeasible integer constrained variable.

1. $variable_index \leftarrow 0$; $infeasibility \leftarrow 0$.
2. for each basic variable x_i do
3. if (the type of x_i restricts it to be integer and the difference of x_i^* from its nearest integer is larger than $infeasibility$) then
4. $variable_index \leftarrow i$. $infeasibility \leftarrow$ the difference of x_i^* from its nearest integer.
5. extract the row of the current simplex tableau corresponding to $variable_index$.
6. generate the Gomory cut from that row.
7. return the Gomory cut.

Procedure Add_Gomory_Cut

INPUT: the linear programming relaxation $\mathcal{L}(P^+)$ of P with the current basis, and the current Gomory cut $\alpha x \geq \beta$.

OUTPUT: new formulation of $\mathcal{L}(P^+)$ with one extra row for the added Gomory cut keeping the current basic solution unchanged.

1. express the Gomory cut $\alpha x \geq \beta$ in terms of the current nonbasic variables.
2. consider $\alpha' x' \geq \beta'$ is the Gomory cut in terms of the current nonbasic variables.
3. introduce a slack variable, say x_{n+m+1}, to make the cut $\alpha' x' - x_{n+m+1} = \beta'$.
4. add $x_{n+m+1} = -\beta' + \alpha' x'$ as a new row to the $\mathcal{L}(P^+)$.
5. return $\mathcal{L}(P^+)$.

Procedure PGC$_0$_Type1_Pivot

INPUT: the linear programming relaxation $\mathcal{L}(P^+)$ of P with the current basis, the current basic feasible solution x^*.

OUTPUT: return *true* if it finds a PC Type 1 pivot, else return *false*.

1. $p \leftarrow 0$; $q \leftarrow 0$; $existP1 \leftarrow 0$.
2. for each nonbasic variable x_j do
3. if (the type of x_j does not restrict it to be integer) then
4. maintaining primal feasibility, find the basic variable x_i which should leave the basis if x_j enters into the basis.
5. if (the type of x_i restricts it to be integer) then
6. evaluate the new basic feasible solution x^*_{new} if this pivot between x_i and x_j is performed.
7. $newObjective \leftarrow$ objective value at the solution x^*_{new}.
8. if ($existP1 = 0$) then
9. $p \leftarrow i$; $q \leftarrow j$; $objective \leftarrow newObjective$; $existP1 \leftarrow 1$.
10. elsif ($newObjective < objective$) then
11. $p \leftarrow i$; $q \leftarrow j$; $objective \leftarrow newObjective$.
12. if ($existP1 \neq 0$) then
13. perform the pivot between x_p and x_q.
14. return *true*.
15. return *false*.

Procedure PGC_0_Type2_Pivot

INPUT: the linear programming relaxation $\mathcal{L}(P^+)$ of P with the current basis, the current basic feasible solution x^*.

OUTPUT: return *true* if it finds a PC Type 2 pivot, else return *false*.

1. $currentIntegerInfeasibility \leftarrow$ sum of the measure of integer infeasibility at the current solution x^*.
2. for each nonbasic variable x_j do
3. maintaining primal feasibility, find the basic variable x_i which should leave the basis if x_j enters into the basis.
4. if (the types of x_i and x_j both restrict them to be integer or both do not restrict them to be integer) then
5. evaluate the new basic feasible solution x^*_{new} if this pivot between x_i and x_j is performed.
6. $newIntegerInfeasibility \leftarrow$ sum of the measure of integer infeasibility at the solution x^*_{new}.
7. if ($newIntegerInfeasibility < currentIntegerInfeasibility$) then
8. perform the pivot between x_i and x_j.
9. return *true*.
10. return *false*.

Procedure PGC_1_Type1_Pivot

INPUT: the linear programming relaxation $\mathcal{L}(P^+)$ of P with the current basis, the current basic feasible solution x^*, the current Gomory cut $\alpha x \geq \beta$.

OUTPUT: return *true* if it finds a PGC_1 Type 1 pivot, else return *false*.

1. $current_lhs \leftarrow \alpha x$ evaluated at the current solution x^*.
2. for each nonbasic variable x_j do
3. if (the type of x_j does not restrict it to be integer) then
4. maintaining primal feasibility, find the basic variable x_i which should leave the basis if x_j enters into the basis.
5. if (the type of x_i restricts it to be integer) then
6. evaluate the new basic feasible solution x^*_{new} if this pivot between x_i and x_j is performed.
7. $new_lhs \leftarrow \alpha x$ evaluated at the new solution x^*_{new}.
8. if ($(\beta - new_lhs) < (\beta - current_lhs)$) then
9. perform the pivot between x_i and x_j.
10. return *true*.
11. return *false*.

Procedure PGC_1_Type2_Pivot

INPUT: the linear programming relaxation $\mathcal{L}(P^+)$ of P with the current basis, the current basic feasible solution x^*, the current Gomory cut $\alpha x \geq \beta$.

OUTPUT: return *true* if it finds a PGC_1 Type 2 pivot, else return *false*.

1. $currentDifference \leftarrow \beta - (\alpha x$ evaluated at the current solution $x^*)$.
2. $p \leftarrow 0$; $q \leftarrow 0$; $countEligibleP2 \leftarrow 0$.
3. for each nonbasic variable x_j do
4. maintaining primal feasibility, find the basic variable x_i which should leave the basis if x_j enters into the basis.
5. if (the types of x_i and x_j both restrict them to be integer or both do not restrict them to be integer) then
6. evaluate the new basic feasible solution x^*_{new} if this pivot between x_i and x_j is performed.
7. $newDifference \leftarrow \beta - (\alpha x$ evaluated at the solution $x^*_{new})$.
8. if ($newDifference < currentDifference$) then
9. $countEligibleP2 \leftarrow countEligibleP2 + 1$.
10. if ($countEligibleP2 \leq \lfloor \log n \rfloor$) then
11. if ($countEligibleP2 = 1$) then
12. $p \leftarrow i$; $q \leftarrow j$; $difference \leftarrow newDifference$.
13. elsif (($newDifference < difference$) or ($newDifference < 0$ and $newDifference > difference$)) then
14. $p \leftarrow i$; $q \leftarrow j$; $difference \leftarrow newDifference$.
15. else exit from the for loop.
16. if ($countEligibleP2 \neq 0$) then
17. perform the pivot between x_p and x_q.
18. return *true*.
19. return *false*.

Procedure Restart

INPUT: the linear programming relaxation $\mathcal{L}(P^+)$ of P with the current basis,
and the current Gomory cut $\alpha x \geq \beta$.

OUTPUT: if successful return *true*, else return *false*.

1. *status* $\leftarrow$ *false*.
2. express the Gomory cut $\alpha x \geq \beta$ in terms of the current nonbasic variables.
3. consider $\alpha' x' \geq \beta'$ is the Gomory cut in terms of the current nonbasic variables.
4. replace the objective function "min cx" of $\mathcal{L}(P^+)$ with "max $\alpha' x'$".
5. repeat
6. choose a nonbasic variable x_j according to the bounded variable simplex method.
7. if (there is no nonbasic variable which should enter the basis) then
8. return *false*.
9. using the bounded variable simplex method, find a basic variable x_i which should leave the basis if x_j enters into the basis.
10. if (there is no basic variable x_i which should leave the basis) then
11. introduce a slack variable, say x_{n+m+1}, to make the cut $\alpha' x' - x_{n+m+1} = \beta'$.
12. add $x_{n+m+1} = -\beta' + \alpha' x'$ as a new row to the $\mathcal{L}(P^+)$.
13. perform a pivot between x_j and x_{n+m+1} to make the basic solution feasible.
14. re-establish the objective function of $\mathcal{L}(P^+)$ to "min cx".
15. *status* $\leftarrow$*true*.
16. else
17. perform pivot between x_i and x_j.
18. if (αx evaluated at new solution is greater or equal to β) then
19. introduce a slack variable, say x_{n+m+1}, to make the cut $\alpha' x' - x_{n+m+1} = \beta'$.
20. add $x_{n+m+1} = -\beta' + \alpha' x'$ as a new row to the $\mathcal{L}(P^+)$.
21. re-establish the objective function of $\mathcal{L}(P^+)$ to "min cx".
22. *status* $\leftarrow$*true*.
23. until *status*
24. return *status*.

Algorithm PGC_0
INPUT: a 0-1 mixed integer problem P and a time limit T.
OUTPUT: a P-feasible solution x^* (null in case of failure).

```
1.  x* ← null, elapsedTime ← 0
2.  construct L(P+) from P
3.  find optimal solution x* of L(P+) using bounded variable revised simplex algorithm
4.  if (Check_P_Feasible) then return x*.
5.  elsif (Round_Truncate is not null) then
        return the P-feasible solution obtained from the procedure Round_Truncate.
6.  repeat
    BEGIN SEARCH PHASE
7.      Generate_Gomory_Cut.
8.      atDeadEnd ← false
9.      repeat
10.         while (PGC0_Type1_Pivot)
11.             x* ← resulting L(P+)-feasible solution
12.             if (x* satisfies αx ≥ β) then
13.                 if (Check_P_Feasible)) then return x*
14.                 else
15.                     Add_Gomory_Cut.
16.                     Generate_Gomory_Cut.
17.         if (PGC0_Type2_Pivot) then
18.             x* ← resulting L(P+)-feasible solution
19.             if (x* satisfies αx ≥ β) then
20.                 if (Check_P_Feasible) then return x*
21.                 else
22.                     Add_Gomory_Cut.
23.                     Generate_Gomory_Cut.
24.         else atDeadEnd ← true
25.     until atDeadEnd
    END SEARCH PHASE
26.     if (Round_Truncate is not null) then
            return the P-feasible solution obtained from the procedure Round_Truncate.
    BEGIN RESTART PHASE
27.     if (not Restart) then return null.
    END RESTART PHASE
28. until (elapsedTime≥ T)
29. return x*
```

Algorithm PGC_1
INPUT: a 0-1 mixed integer problem P and a time limit T.
OUTPUT: a P-feasible solution x^* (null in case of failure).

```
1.  x* ← null, elapsedTime ← 0
2.  construct L(P+) from P
3.  find optimal solution x* of L(P+) using bounded variable revised simplex algorithm
4.  if (Check_P_Feasible) then return x*.
5.  elsif (Round_Truncate is not null) then
        return the P-feasible solution obtained from the procedure Round_Truncate.
6.  repeat
    BEGIN SEARCH PHASE
7.      Generate_Gomory_Cut.
8.      atDeadEnd ← false
9.      repeat
10.         while (PGC1_Type1_Pivot)
11.             x* ← resulting L(P+)-feasible solution
12.             if (x* satisfies αx ≥ β) then
13.                 if (Check_P_Feasible)) then return x*
14.                 else
15.                     Add_Gomory_Cut.
16.                     Generate_Gomory_Cut.
17.         if (PGC1_Type2_Pivot) then
18.             x* ← resulting L(P+)-feasible solution
19.             if (x* satisfies αx ≥ β) then
20.                 if (Check_P_Feasible) then return x*
21.                 else
22.                     Add_Gomory_Cut.
23.                     Generate_Gomory_Cut.
24.         else atDeadEnd ← true
25.     until atDeadEnd
    END SEARCH PHASE
26.     if (Round_Truncate is not null) then
            return the P-feasible solution obtained from the procedure Round_Truncate.
    BEGIN RESTART PHASE
27.     if (not Restart) then return null.
    END RESTART PHASE
28. until (elapsedTime≥ T)
29. return x*
```

Appendix B

Experimental Results

B.1 Benchmark Instances

We present the description of the benchmark instances in the following tables.

Table B.1: All mixed integer program instances from MIPLIB 2003.

Name	Constraints	Variables	Binary variables	Integer variables	Objective value of Optimal/Best known* solution
10teams	230	2025	1800	0	924
a1c1s1	3312	3648	192	0	11505.43*
aflow30a	479	842	421	0	1158
aflow40b	1442	2728	1364	0	1168
air04	823	8904	8904	0	56137
air05	426	7195	7195	0	26374
arki001	1048	1388	415	123	7580813.04*
atlanta-ip	21732	48738	46667	106	95.009*
cap6000	2176	6000	6000	0	−2451377
dano3mip	3202	13873	552	0	688.26*
danoint	664	521	56	0	65.6667

Continued on next page

Table B.1 – Continued from previous page

Name	Constraints	Variables	Binary variables	Integer variables	Objective value of Optimal/Best known* solution
ds	656	67732	67732	0	413.78*
disctom	399	10000	10000	0	−5000
fast0507	507	63009	63009	0	174
fiber	363	1298	1254	0	405935
fixnet6	478	878	378	0	3983
gesa2	1392	1224	240	168	2.57e+07
gesa2-o	1248	1224	384	336	2.57e+07
glass4	396	322	302	0	1460013800.0*
harp2	112	2993	2993	0	−73899798
liu	2178	1156	1089	0	1212*
manna81	6480	3321	18	3303	−13164
markshare1	6	62	50	0	1
markshare2	7	74	60	0	1
mas74	13	151	150	0	11801.2
mas76	12	151	150	0	40005.1
misc07	212	260	259	0	2810
mkc	3411	5325	5323	0	−563.846
mod011	4480	10958	96	0	−5.4558e+07
modglob	291	422	98	0	2.07405e+07
momentum1	42680	5174	2249	0	346535*
momentum2	24237	3732	1808	1	15216.987*
momentum3	56882	13532	6598	1	370177.036*
msc98-ip	15850	21143	20237	53	2.327e+07*
mzzv11	9499	10240	9989	251	−21718
mzzv42z	10460	11717	11482	235	−20540
net12	14021	14115	1603	0	214
noswot	182	128	75	25	−41
nsrand-ipx	735	6621	6620	0	51200

Continued on next page

Table B.1 – Continued from previous page

Name	Constraints	Variables	Binary variables	Integer variables	Objective value of Optimal/Best known* solution
nw04	36	87842	87842	0	16862
opt1217	64	769	768	0	−16
p2756	755	2756	2756	0	3124
pk1	45	86	55	0	11
pp08a	136	240	64	0	7350
pp08aCUTS	246	240	64	0	7350
protfold	2112	1835	1835	0	-30*
qiu	1192	840	48	0	−132.873
rd-rplusc-21	125899	622	457	0	167297.61*
roll3000	2295	1166	246	492	12890*
rout	291	556	300	15	1077.56
set1ch	492	712	240	0	54537.8
seymor	4945	1372	1372	0	423
sp97ar	1761	14101	14101	0	661778926.6*
stp3d	159488	204880	204880	0	No solution known
swath	884	6805	6724	0	471.03*
t1717	551	73885	73885	0	216557*
timtab1	171	397	64	107	764772
timtab2	294	675	113	181	1096557*
tr12-30	750	1080	360	0	130596
vpm2	234	378	168	0	13.75

Table B.2: All mixed integer program instances from DEIS operations research library.

Name	Constraints	Variables	Binary variables	Integer variables	Objective value of Optimal/Best known* solution
a1c1s1	3312	3648	192	0	11505.43*
a2c1s1	3312	3648	192	0	10889.14*
b1c1s1	3904	3872	288	0	24544.25*
b2c1s1	3904	3872	288	0	25740.15*
biella1	1203	7328	6110	0	3065084.57*
nsr8k	6284	38356	32040	0	2.0301e+07*
rail507	509	63019	63009	0	174*
rail2536c	2539	15293	15284	0	689*
rail2586c	2589	13226	13215	0	953*
rail4284c	4287	21714	21705	0	1071*
rail4872c	4875	24656	24645	0	1550*
sp97ar	1761	14101	14101	0	661778926.6*
sp97ic	1034	12497	12497	0	428079014.2*
sp98ar	1436	15085	15085	0	529814784.7*
sp98ic	826	10894	10894	0	449144758.4*
bg512142	1307	792	240	0	189183.16*
dg012142	6310	2080	640	0	2706923.5*
dc1c	1649	10039	8380	0	1843531.06*
dc1l	1653	37297	35638	0	1813853.08*
dolom1	1803	11612	9720	0	20560415.03*
siena1	2220	13741	11775	0	13360676.13*
trento1	1265	7687	6415	0	5190144*
CMS750_4	16381	11697	7196	0	253*
berlin_5_8_0	1532	1083	794	0	62*
railway_8_1_0	2527	1796	1177	0	400*
usAbbrv.8.25_70	3291	2312	1681	0	121*

Continued on next page

Table B.2 – Continued from previous page					
Name	Constraints	Variables	Binary variables	Integer variables	Objective value of Optimal/Best known* solution
blp-ic97	923	9845	9753	0	4048.35*
blp-ic98	717	13640	13550	0	4494.68*
blp-ar98	1128	16021	15806	0	6211.45*
blp-ir98	486	6097	6031	0	2342.31*
umts	4465	2947	2802	72	30121483*

Table B.3: 15 new 0-1 mixed integer program instances used in [29].

Name	Constraints	Variables	Binary variables	Integer variables	Objective value of Optimal/Best known* solution
ljb2	1482	771	681	0	0.5077*
ljb7	8133	4163	3920	0	0.1145*
ljb9	9231	4721	4460	0	0.739*
ljb10	10742	5496	5196	0	0.508*
ljb12	9596	4913	4633	0	0.399*
rococoB10-011000	1667	4456	4320	136	19449*
rococoB10-011001	1677	4456	4320	136	21265*
rococoB11-010000	3792	12376	12210	166	32246*
rococoB11-110001	8148	12431	12265	166	42444*
rococoB12-111111	8978	9109	8778	331	39831*
rococoC10-001000	1293	3117	2993	124	11460*
rococoC10-100001	7596	5864	5740	124	16664*
rococoC11-010100	4010	12321	12155	166	20889*
rococoC11-011100	2367	6491	6325	166	20889*
rococoC12-100000	21550	17299	17112	187	35512*
rococoC12-111100	10842	8619	8432	187	35909*

B.2 PGC Experimental Results

We present the details of experimental results related to PGC in the following tables.

Table B.4: Experimental results of PC, PGC_0, and PGC_1 on 77 benchmark instances. + : a time limit of 1 CPU-hour exceeded. − : heuristic reported failure. Solution: objective value of found solution. Time: seconds to find a solution or to report failure excluding the time to solve the initial LP. R: the number of times restarted.

Name	PC			PGC_0			PGC_1		
	Solution	Time	R	Solution	Time	R	Solution	Time	R
10teams	−	1.2	1	−	+	379	−	+	197
a1c1s1	−	95.1	1	20439.08	231.4	9	21987.69	92.4	2
aflow30a	−	0.6	1	2136	0.6	1	1930	0.2	0
aflow40b	−	7.1	1	2177	15.6	5	2719	5.5	3
air04	−	+	1	56913	41.1	0	57974	32.3	3
air05	−	12.9	1	34442	88.3	13	30174	16.3	3
cap6000	-2407434	4.7	0	-2442801	0.1	0	−2442801	0.1	0
dano3mip	−	+	0	−	+	0	−	+	0
danoint	−	9.2	1	66.5	17.0	0	66.5	170.4	0
ds	−	88.1	1	−	+	35	−	+	41
disctom	−	26.9	1	−5000	437.9	41	−5000	927.8	47
fast0507	250	32.1	0	239	36.0	0	282	52.4	3
fiber	2131825.48	1.4	0	2131825.48	1.4	0	2323978.76	0.6	0
fixnet6	59044	6.4	0	63144	6.5	0	92308	0.8	0
glass4	−	0.2	1	12000171916	0.7	17	3000029600	3.1	11
harp2	−	0.4	1	-61991165	3.4	42	−62747758	1.8	17
liu	3998	22.1	0	6450	0.1	0	6450	0.1	0
markshare1	340	0.0	0	923	0.0	0	923	0.0	0

Continued on next page

Table B.4 – Continued from previous page

Name	PC			PGC_0			PGC_1		
	Solution	Time	R	Solution	Time	R	Solution	Time	R
markshare2	558	0.0	0	558	0.0	0	558	0.0	0
mas74	19197.46	0.0	0	56379.25	0.0	0	56379.25	0.0	0
mas76	44839.82	0.0	0	80297.61	0.0	0	80297.61	0.0	0
misc07	–	0.2	1	4815	6.0	148	4445	1.2	36
mkc	-2.05	18.4	0	-2.05	18.7	0	335.15	0.2	0
mod011	−42783986	78.6	0	0.0	0.1	0	0.0	0.1	0
modglob	20808306.8	0.3	0	35147088.8	0.0	0	35147088.8	0.0	0
momentum1	–	+	0	–	+	0	–	+	0
net12	–	2266.3	1	–	+	4	–	+	79
nsrand-ipx	237600	0.4	0	237600	0.7	0	336000	0.4	3
nw04	19882	1.8	0	19882	4.1	0	17516	2.1	0
opt1217	−16	0.0	0	−14	0.0	0	−14	0.0	0
p2756	–	1.8	1	119888	4.1	24	111375	3.3	1
pk1	35	0.0	0	48	0.0	0	48	0.0	0
pp08a	11270	0.1	0	11180	0.1	0	17700	0.0	2
pp08aCUTS	9950	0.2	0	9950	0.2	0	13670	0.1	0
protfold	–	219.7	1	–	+	255	–	+	99
qiu	–	1.1	1	2728.93	185.2	44	706.34	118.8	27
rd-rplusc-21	–	554.5	1	–	+	4	–	+	4
set1ch	101342	3.4	0	101342	3.4	0	170306	1.0	0
seymor	616	134.3	0	617	133.9	0	644	37.8	7
sp97ar	847665971	9.9	0	847665971.4	9.9	0	885031424.4	12.4	0
swath	–	3.5	1	–	+	259	–	+	243
t1717	–	323.3	1	–	+	9	–	+	4
tr12-30	–	11.0	1	147784	12.2	6	225057	10.6	0
vpm2	–	0.1	0	19.25	0.1	3	21.25	0.1	4
a2c1s1	–	86.8	1	19435.61	125.0	7	21550.91	123.7	4
b1c1s1	–	171.7	1	72509.89	328.4	22	83070.85	545.7	7
b2c1s1	–	228.8	1	65868.57	1203	8	66650.90	386.0	2

Continued on next page

Table B.4 – Continued from previous page

Name	PC			PGC_0			PGC_1		
	Solution	Time	R	Solution	Time	R	Solution	Time	R
biella1	–	154.1	1	115610810.1	204.7	2	25976128.6	117.6	8
nsr8k	–	+	0	–	+	0	–	+	0
rail507	248	52.3	0	241	54.8	0	278	29.8	0
rail2536c	975	320.3	0	788	320.4	0	942	43.0	2
rail2586c	1491	1008.7	0	1531	996.4	0	1617	469.6	3
rail4284c	–	+	0	–	+	0	–	+	0
rail4872c	–	+	0	–	+	0	–	+	0
sp97ic	660936153	1.4	0	678371143.0	1.7	0	604080493.7	2.0	0
sp98ar	–	16.9	1	737228733.6	7.7	0	785793816.3	6.6	0
sp98ic	614965469	1.1	0	550312567.3	1.5	0	641234440.3	2.4	1
bg512142	8774550	30.7	0	120738665	0.0	0	120738665	0.0	0
dg012142	34067266	923.2	0	153406945.5	0.1	0	153406945.5	0.1	0
dc1c	–	481.3	1	178531140.7	446.0	1	9999999999.9	272.3	8
dc1l	–	1633.8	1	150135254.2	1038	0	162070107.1	1127	3
dolom1	–	+	0	–	+	0	–	+	11
siena1	–	+	0	–	+	0	–	+	0
trento1	–	374.5	1	437090154.0	432.9	0	10000000000	131.8	2
cms750-4	–	+	0	–	+	0	–	+	0
berlin-5-8-0	–	25.5	1	80	35.4	20	79	22.0	20
railway-8-1-0	–	90.1	1	441	125.3	50	440	77.6	21
usabbrv-8-25-70	–	272.9	1	165	361.4	52	160	180.9	34
blp-ic97	–	6.6	1	6791.58	11.0	6	5953.77	17.7	9
blp-ic98	–	10.8	1	10089.33	25.4	15	7983.30	20.8	6
blp-ar98	–	25.5	1	11472.17	28.6	5	10670.70	49.7	7
blp-ir98	–	4.4	1	2930.49	5.6	8	3688.45	4.1	4
ljb2	0.976	22.5	0	7.23	0.1	0	7.23	0.1	0
ljb7	–	+	0	8.61	1.9	0	8.61	1.9	0
ljb9	–	+	0	9.47	2.3	0	9.47	2.3	0
ljb10	–	+	0	7.31	2.7	0	7.31	2.7	0

Continued on next page

Table B.4 – Continued from previous page

Name	PC			PGC_0			PGC_1		
	Solution	Time	R	Solution	Time	R	Solution	Time	R
ljb12	–	+	0	6.19	1.9	0	6.19	1.9	0

Table B.5: Experimental results of FP and PGC_1 on 77 benchmark instances. + : a time limit of 1 CPU-hour exceeded. – : heuristic reported failure. Solution: objective value of found solution. Time: seconds to find a solution or to report failure excluding the time to solve the initial LP. N: the number of iterations.

Name	FP			PGC_1	
	Solution	Time	N	Solution	Time
10teams	1024	13.9	49	–	+
a1c1s1	19280.05	15.6	6	21987.69	92.4
aflow30a	4351	0.3	5	1930	0.2
aflow40b	6035	0.7	4	2719	5.5
air04	60175	333.4	10	57974	32.3
air05	31988	12.1	2	30174	16.3
cap6000	-2322939	0.8	7	−2442801	0.1
dano3mip	813.105	201.0	3	–	+
danoint	76.33	2.1	12	66.5	170.4
ds	–	+	24	–	+
disctom	-5000	56.1	3	−5000	927.8
fast0507	204	282.8	3	282	52.4
fiber	2270781.03	0.5	5	2323978.76	0.6
fixnet6	14665	0.1	5	92308	0.8
glass4	13200143054	39.0	3108	3000029600	3.1
harp2	-39771742	2.4	116	−62747758	1.8
liu	6022	0.5	0	6450	0.1

Continued on next page

Table B.5 – Continued from previous page

Name	FP			PGC_1	
	Solution	Time	N	Solution	Time
markshare1	1329	0.0	5	923	0.0
markshare2	558	0.0	2	558	0.0
mas74	14372.87	0.0	2	56379.25	0.0
mas76	43774.25	0.0	2	80297.61	0.0
misc07	3945	1.7	56	4445	1.2
mkc	-85.85	0.6	3	335.15	0.2
mod011	0.0	0.1	0	0.0	0.1
modglob	35147088.88	0.0	0	35147088.88	0.0
momentum1	–	+	20	–	+
net12	–	+	81	–	+
nsrand-ipx	357920	1.0	3	336000	0.4
nw04	19792	3.2	1	17516	2.1
opt1217	0	0.0	0	−14	0.0
p2756	–	+	12993	111375	3.3
pk1	48	0.0	0	48	0.0
pp08a	13190	0.0	4	17700	0.0
pp08aCUTS	12280	0.1	3	13670	0.1
protfold	–	+	34	–	+
qiu	2416.85	0.6	3	706.34	118.8
rd-rplusc-21	–	+	14	–	+
set1ch	149959	0.1	11	170306	1.0
seymor	481	35.4	4	644	37.8
sp97ar	1561625924	27.5	4	885031424	12.4
swath	30965.35	10.3	57	–	+
t1717	–	+	13	–	+
tr12-30	271121	3.0	21	225057	10.6
vpm2	31.5	0.1	6	21.25	0.1
a2c1s1	24016.87	121.0	48	21550.91	123.7
b1c1s1	68768.09	29.0	9	83070.85	545.7

Continued on next page

Table B.5 – Continued from previous page

Name	FP			PGC_1	
	Solution	Time	N	Solution	Time
b2c1s1	67760.77	36.5	9	66650.90	386.0
biella1	9999999999	29.5	4	25976128.64	117.6
nsr8k	–	+	2	–	+
rail507	181	249.5	3	278	29.8
rail2536c	–	+	6	942	43.0
rail2586c	10000000000	1321	3	1617	469.6
rail4284c	–	+	4	–	+
rail4872c	–	+	3	–	+
sp97ic	723692147	10.3	4	604080493	2.0
sp98ar	966779165	42.5	6	785793816	6.6
sp98ic	1200994883	7.0	4	641234440	2.4
bg512142	120738665	0.2	0	120738665	0.0
dg012142	153406945.5	1.4	0	153406945.5	0.1
dc1c	9999999999	222.5	6	9999999999	272.3
dc1l	26899359.89	1195	6	162070107.1	1127
dolom1	10000000000	637.0	6	–	+
siena1	–	+	3	–	+
trento1	10000000000	145.1	2	10000000000	131.8
cms750-4	–	+	33	–	+
berlin-5-8-0	80	4.0	15	79	22.0
railway-8-1-0	441	11.7	13	440	77.6
usabbrv-8-25-70	172	96.8	80	160	180.9
blp-ic97	8504.96	4.4	10	5953.77	17.7
blp-ic98	13973.87	4.0	7	77983.30	20.8
blp-ar98	–	+	1648	10670.70	49.7
blp-ir98	6866.19	1.4	7	3688.45	4.1
ljb2	7.23	0.5	0	7.23	0.1
ljb7	8.61	13.4	0	8.61	1.9
ljb9	9.47	18.3	0	9.47	2.3

Continued on next page

Table B.5 – Continued from previous page					
Name	FP			PGC_1	
	Solution	Time	N	Solution	Time
ljb10	7.31	28.1	0	7.31	2.7
ljb12	6.19	19.1	0	6.19	1.9

Table B.6: Time to solve the initial linear programming relaxation of 77 benchmark instances by GLPK 4.0 and Cplex 9.13.

Name	LP solution time (secs)		Name	LP solution time (secs)	
	GLPK	Cplex		GLPK	Cplex
10teams	0.18	0.09	sp97ar	8.70	1.93
a1c1s1	1.21	0.01	swath	0.09	0.08
aflow30a	0.03	0.03	t1717	84.09	8.57
aflow40b	0.29	0.09	tr12-30	0.12	0.00
air04	14.42	2.55	vpm2	0.01	0.00
air05	2.48	0.44	a2c1s1	1.16	0.04
cap6000	0.43	0.09	b1c1s1	1.96	0.04
dano3mip	55.14	52.97	b2c1s1	2.04	0.05
danoint	0.15	0.06	biella1	8.54	4.56
ds	83.97	44.07	nsr8k	2903.14	962.27
disctom	15.63	4.05	rail507	46.78	23.32
fast0507	48.01	21.53	rail2536c	101.57	9.50
fiber	0.02	0.00	rail2586c	116.91	37.81
fixnet6	0.02	0.00	rail4284c	496.63	135.97
glass4	0.00	0.00	rail4872c	527.94	167.22
harp2	0.04	0.03	sp97ic	3.31	1.37
liu	0.21	0.02	sp98ar	7.98	2.49
markshare1	0.00	0.00	sp98ic	2.99	1.24
Continued on next page					

Table B.6 – Continued from previous page

Name	LP solution time (secs)		Name	LP solution time (secs)	
	GLPK	Cplex		GLPK	Cplex
markshare2	0.00	0.00	bg512142	0.23	0.09
mas74	0.00	0.00	dg012142	1.63	0.35
mas76	0.00	0.00	dc1c	21.00	10.30
misc07	0.02	0.00	dc1l	145.78	52.02
mkc	0.24	0.10	dolom1	39.88	18.38
mod011	0.96	0.13	siena1	129.09	49.18
modglob	0.01	0.00	trento1	32.18	11.31
momentum1	49.43	1.36	cms750-4	57.16	0.89
net12	4.69	6.35	berlin-5-8-0	0.27	0.02
nsrand-ipx	0.29	0.28	railway-8-1-0	0.93	0.06
nw04	3.17	2.16	usabbrv-8-25-70	1.49	0.10
opt1217	0.01	0.01	blp-ic97	0.45	0.29
p2756	0.01	0.00	blp-ic98	0.66	0.45
pk1	0.00	0.00	blp-ar98	1.41	0.54
pp08a	0.01	0.00	blp-ir98	0.18	0.15
pp08aCUTS	0.01	0.01	ljb2	0.21	0.02
protfold	3.08	1.10	ljb7	9.09	0.57
qiu	0.26	0.09	ljb9	12.80	0.74
rd-rplusc-21	136.26	2.09	ljb10	17.05	1.01
set1ch	0.01	0.01	ljb12	13.30	0.76
seymor	5.21	1.70			

Table B.7: Experimental results of Cplex-D, Cplex-F, and PGC_1 on 77 benchmark instances. + : a time limit of 1 CPU-hour exceeded. − : heuristic reported failure. Solution: objective value of found solution. Time: seconds to find a solution or to report failure excluding the time to solve the initial LP. Nd: the number of nodes of the search tree.

Name	Cplex-D			Cplex-F			PGC_1	
	Solution	Time	Nd	Solution	Time	Nd	Solution	Time
10teams	952	3.4	0	924	6.1	385	−	+
a1c1s1	14595.12	4.9	0	13174.27	2.7	140	21987.69	92.4
aflow30a	1307	0.1	0	1307	0.1	0	1930	0.2
aflow40b	1489	12.3	550	1635	26.5	3480	2719	5.5
air04	57306	2.0	0	59734	4.7	43	57974	32.3
air05	26861	0.9	0	31189	2.4	90	30174	16.3
cap6000	−2445344	0.2	0	−2445344	0.2	0	−2442801	0.1
dano3mip	768.375	8.5	0	768.375	8.4	0	−	+
danoint	66.5	0.6	0	69.5	1.2	51	66.5	170.4
ds	5418.56	3.4	0	5418.56	1.5	0	−	+
disctom	-5000	208.8	61	−5000	83.6	4843	−5000	927.8
fast0507	201	1.6	0	201	1.6	0	282	52.4
fiber	422169.4	0.1	0	422169.4	0.1	0	2323978.76	0.6
fixnet6	4505	0.0	0	4505	0.0	0	92308	0.8
glass4	2.900e+09	0.2	171	3.8167e+09	1.2	5660	3000029600	3.1
harp2	-7.2977e+07	0.1	0	-7.2977e+07	0.1	0	−62747758	1.8
liu	6450	0.1	0	6450	0.1	0	6450	0.1
markshare1	1095	0.0	0	1095	0.0	0	923	0.0
markshare2	944	0.0	0	944	0.0	0	558	0.0
mas74	19197.46	0.0	0	19197.46	0.0	0	56379.25	0.0
mas76	44877.42	0.0	0	44877.42	0.0	0	80297.61	0.0
misc07	3615	0.3	10	3870	0.1	30	4445	1.2
mkc	-496.46	0.2	0	-496.46	0.2	0	335.15	0.2

Continued on next page

Table B.7 – Continued from previous page

Name	Cplex-D			Cplex-F			PGC_1	
	Solution	Time	Nd	Solution	Time	Nd	Solution	Time
mod011	−42902314	0.2	0	−42902314	0.1	0	0.0	0.1
modglob	20786787.0	0.0	0	20786787.0	0.0	0	35147088.8	0.0
momentum1	–	+	100	–	+	500	–	+
net12	214	838.0	130	255	411.1	185	–	+
nsrand-ipx	57600	0.5	0	57600	0.5	0	336000	0.4
nw04	18228	6.0	0	18228	5.9	0	17516	2.1
opt1217	−1	0.0	0	−12	0.1	0	−14	0.0
p2756	3378	0.1	0	3378	0.1	0	111375	3.3
pk1	57	0.0	0	57	0.0	0	48	0.0
pp08a	15300	0.0	0	15300	0.0	0	17700	0.0
pp08aCUTS	13490	0.0	0	13490	0.0	0	13670	0.1
protfold	–	+	3200	−14	506.7	260	–	+
qiu	1805.17	0.1	0	1805.17	0.1	0	706.34	118.8
rd-rplusc-21	–	+	33800	–	+	35800	–	+
set1ch	107267	0.0	0	107267	0.0	0	170306	1.0
seymor	457	0.4	0	457	0.4	0	644	37.8
sp97ar	697232594.6	0.9	0	697232594.6	0.9	0	885031424	12.4
swath	1405.57	0.2	0	1405.57	0.2	0	–	+
t1717	233768	480.1	720	342258	261.8	890	–	+
tr12-30	146513	0.8	0	141874	0.6	90	225057	10.6
vpm2	16	0.0	0	16.75	0.0	0	21.25	0.1
a2c1s1	20865.33	0.1	0	20865.33	0.1	0	21550.91	123.7
b1c1s1	69933.52	0.1	0	69933.52	0.1	0	83070.85	545.7
b2c1s1	70575.52	0.1	0	70575.52	0.1	0	66650.90	386.0
biella1	3154183.34	447.2	240	3377565.2	67.5	570	25976128.6	117.6
nsr8k	–	+	0	–	+	0	–	+
rail507	198	2.0	0	198	2.0	0	278	29.8
rail2536c	762	0.7	0	762	0.6	0	942	43.0
rail2586c	1073	1.0	0	1073	0.9	0	1617	469.6

Continued on next page

Table B.7 – Continued from previous page

Name	Cplex-D			Cplex-F			PGC_1	
	Solution	Time	Nd	Solution	Time	Nd	Solution	Time
rail4284c	1213	1.9	0	1213	1.9	0	–	+
rail4872c	1723	2.8	0	1723	2.8	0	–	+
sp97ic	470971128.6	1.0	0	470971128.6	0.9	0	604080493	2.0
sp98ar	566534466.7	1.3	0	566534466.7	1.2	0	785793816	6.6
sp98ic	513738135	0.9	0	513738135	0.9	0	641234440	2.4
bg512142	120670203.5	0.1	0	120670203.5	0.1	0	120738665	0.0
dg012142	153397324	0.3	0	153397324	0.3	0	153406945.5	0.1
dc1c	19026816.89	2717.9	9459	6044895.95	256.7	4572	9999999999	272.3
dc1l	751870828.7	1.8	0	751870828.7	1.8	0	162070107.1	1127
dolom1	–	+	6900	44243931.13	226.0	315	–	+
siena1	–	+	2100	74142275.93	861.0	590	–	+
trento1	46015409	32.9	0	9076504.046	141.6	210	10000000000	131.8
cms750-4	296	498.9	920	342	18.8	940	–	+
berlin-5-8-0	64	1.5	1470	73	0.3	155	79	22.0
railway-8-1-0	406	2.3	0	418	0.9	326	440	77.6
usabbrv-8-25-70	130	156.7	42290	–	+	80	160	180.9
blp-ic97	4149.34	27.0	360	4295.57	19.0	2200	5953.77	17.7
blp-ic98	4695.86	51.7	1000	4777.00	26.7	1910	77983.30	20.8
blp-ar98	6554.69	112.7	2640	6636.40	50.0	3880	10670.70	49.7
blp-ir98	2364.10	3.1	70	2775.23	2.4	410	3688.45	4.1
ljb2	1.726	0.3	0	1.457	0.2	30	7.23	0.1
ljb7	8.61	2.1	0	0.732	2.3	100	8.61	1.9
ljb9	9.47	3.6	0	2.026	4.4	180	9.47	2.3
ljb10	1.916	2.7	0	1.13	3.3	90	7.31	2.7
ljb12	3.375	0.8	0	1.40	3.4	110	6.19	1.9

Table B.8: Experimental results of PBS4 and PGC_1 on 10 benchmark 0-1 integer program instances. $-$: no solution found. $+$: a time limit of 1 CPU-hour exceeded. Solution: objective value of found solution. Time: seconds to find a solution.

Name	PBS4		PGC_1		Name	PBS4		PGC_1	
	Solution	Time	Solution	Time		Solution	Time	Solution	Time
air04	$-$	$+$	57974	46.7	nw04	62498	5.8	17516	5.3
air05	41859	6.03	30174	18.8	p2756	267435	0.03	111375	3.3
cap6000	-222820	223.95	-2442801	0.6	protfold	-20	0.01	$-$	$+$
disctom	$-$	$+$	-5000	943.4	seymor	1308	0.01	644	43.1
fast0507	122425	73.99	282	63.4	t1717	406325	359.49	$-$	$+$

Table B.9: Time taken to find a feasible solution on the Cornuéjols-Dawande feasibility model instances generated with $n = 10m$. $+$: A time limit of 1 CPU-hour exceeded. CD-Feasxx-y: name of the instance, where xx and y represent n and the number of the instance respectively.

Instance	Time taken (seconds)				
	Cplex-D	Cplex-F	PBS4	FP	PGC_1
CD-Feas10-1	0.00	0.00	0.01	0.01	0.00
CD-Feas10-2	0.01	0.01	0.00	0.00	0.01
CD-Feas10-3	0.01	0.01	0.00	0.00	0.00
CD-Feas10-4	0.01	0.00	0.00	0.04	0.00
CD-Feas10-5	0.00	0.00	0.00	0.02	0.00
CD-Feas20-1	0.04	0.09	0.01	2.83	0.41
CD-Feas20-2	0.01	0.02	0.01	1.39	10.66
CD-Feas20-3	0.01	0.00	0.00	0.46	1.39
CD-Feas20-4	0.01	0.01	0.00	1.75	0.14
CD-Feas20-5	0.00	0.00	0.01	1.68	0.04

Continued on next page

Table B.9 – Continued from previous page					
Instance	Time taken (seconds)				
	Cplex-D	Cplex-F	PBS4	FP	PGC_1
CD-Feas30-1	3.60	2.93	5.54	71.26	10.69
CD-Feas30-2	1.61	1.31	218.74	77.52	38.64
CD-Feas30-3	2.03	6.63	254.63	413.96	787.97
CD-Feas30-4	2.66	2.42	9.42	1609.58	2981.63
CD-Feas30-5	9.79	7.50	51.36	743.55	1298.91
CD-Feas40-1	88.48	21.21	+	+	+
CD-Feas40-2	406.21	150.84	+	+	+
CD-Feas40-3	24.05	14.71	+	+	+
CD-Feas40-4	429.71	255.77	+	+	+
CD-Feas40-5	446.87	446.42	+	+	+
CD-Feas50-1	+	+	+	+	+
CD-Feas50-2	+	+	+	+	+
CD-Feas50-3	+	+	+	+	+
CD-Feas50-4	+	+	+	+	+
CD-Feas50-5	+	+	+	+	+

Table B.10: Time taken to find a feasible solution on constrained market-sharing instances generated with $k = 2.0$. +: a time limit of 1 CPU-hour exceeded. CMSxxx-y: name of the instance, where xxx and y represent n and the number of the instance respectively.

Instance	Time taken (seconds)				
	Cplex-D	Cplex-F	PBS4	FP	PGC_1
CMS50-1	+	2.95	3.01	0.90	0.11
CMS50-2	+	1.65	5.13	0.24	0.02
CMS50-3	+	4.34	3.12	3.41	0.05
CMS50-4	+	7.00	+	5.55	0.12
Continued on next page					

Table B.10 – Continued from previous page

Instance	Time taken (seconds)				
	Cplex-D	Cplex-F	PBS4	FP	PGC_1
CMS50-5	+	4.42	7.96	2.30	0.10
CMS100-1	+	1217.24	+	1628.33	1.54
CMS100-2	+	228.87	+	161.31	0.36
CMS100-3	+	776.17	+	25.06	0.87
CMS100-4	+	197.40	+	377.64	5.00
CMS100-5	+	826.36	+	24.73	2.51
CMS150-1	+	+	+	+	11.68
CMS150-2	+	+	+	259.79	104.37
CMS150-3	+	+	+	139.23	1.49
CMS150-4	+	+	+	1088.52	10.23
CMS150-5	+	+	+	+	11.93
CMS200-1	+	+	+	+	399.07
CMS200-2	+	+	+	+	180.00
CMS200-3	+	+	+	+	226.12
CMS200-4	+	+	+	+	44.30
CMS200-5	+	+	+	+	88.78
CMS250-1	+	+	+	+	638.70
CMS250-2	+	+	+	+	+
CMS250-3	+	+	+	+	+
CMS250-4	+	+	+	+	1159.38
CMS250-5	+	+	+	+	466.62
CMS300-1	+	+	+	+	2788.31
CMS300-2	+	+	+	+	2542.34
CMS300-3	+	+	+	+	+
CMS300-4	+	+	+	+	2334.55
CMS300-5	+	+	+	+	1050.26
CMS350-1	+	+	+	+	1684.35
CMS350-2	+	+	+	+	+
CMS350-3	+	+	+	+	+
CMS350-4	+	+	+	+	+

Continued on next page

Table B.10 – Continued from previous page					
Instance	Time taken (seconds)				
	Cplex-D	Cplex-F	PBS4	FP	PGC_1
CMS350-5	+	+	+	+	2540.84
CMS400-1	+	+	+	+	+
CMS400-2	+	+	+	+	+
CMS400-3	+	+	+	+	+
CMS400-4	+	+	+	+	+
CMS400-5	+	+	+	+	+

Table B.11: Time taken to find a feasible solution on constrained market-sharing instances generated with $k = 1.5$. +: a time limit of 1 CPU-hour exceeded. CMSxxx-y: name of the instance, where xxx and y represent n and the number of the instance respectively.

Instance	Time taken (seconds)				
	Cplex-D	Cplex-F	PBS4	FP	PGC_1
CMS50-1	+	43.08	+	523.33	77.89
CMS50-2	+	3.57	+	5.31	0.18
CMS50-3	+	5.56	+	11.62	0.03
CMS50-4	+	15.90	+	68.11	0.12
CMS50-5	+	40.96	+	25.30	23.52
CMS75-1	+	1811.92	+	+	834.04
CMS75-2	+	1582.52	+	832.26	41.27
CMS75-3	+	344.70	+	3329.37	6.66
CMS75-4	+	2094.63	+	2320.46	19.75
CMS75-5	+	124.88	+	541.85	2.32
CMS100-1	+	1198.96	+	+	11.97
CMS100-2	+	+	+	+	282.44
CMS100-3	+	+	+	+	164.05
Continued on next page					

Table B.11 – Continued from previous page

Instance	Time taken (seconds)				
	Cplex-D	Cplex-F	PBS4	FP	PGC_1
CMS100-4	+	3470.68	+	+	29.36
CMS100-5	+	+	+	+	1182.79
CMS125-1	+	+	+	+	746.46
CMS125-2	+	+	+	+	+
CMS125-3	+	+	+	+	388.88
CMS125-4	+	+	+	+	513.99
CMS125-5	+	+	+	+	300.03
CMS150-1	+	+	+	+	+
CMS150-2	+	+	+	+	336.80
CMS150-3	+	+	+	+	+
CMS150-4	+	+	+	+	+
CMS150-5	+	+	+	+	1202.67
CMS175-1	+	+	+	+	+
CMS175-2	+	+	+	+	+
CMS175-3	+	+	+	+	+
CMS175-4	+	+	+	+	+
CMS175-5	+	+	+	+	1875.09
CMS200-1	+	+	+	+	+
CMS200-2	+	+	+	+	+
CMS200-3	+	+	+	+	+
CMS200-4	+	+	+	+	+
CMS200-5	+	+	+	+	+

Table B.12: Time taken to find a feasible solution for constrained market-sharing instances with $k = 1.3$. +: a time limit of 1 CPU-hour exceeded. CMSxxx-y: name of the instance, where xxx and y represent n and the number of the instance respectively.

Instance	Time taken (seconds)				
	Cplex-D	Cplex-F	PBS4	FP	PGC_1
CMS50-1	+	78.75	+	299.60	30.72
CMS50-2	+	36.62	+	1692.57	1.79
CMS50-3	+	54.38	+	125.35	125.41
CMS50-4	+	80.20	+	1403.09	14.75
CMS50-5	+	13.50	+	160.87	0.35
CMS75-1	+	+	+	+	+
CMS75-2	+	+	+	+	+
CMS75-3	+	1841.53	+	+	62.28
CMS75-4	+	+	+	+	436.24
CMS75-5	+	+	+	+	238.98
CMS100-1	+	+	+	+	+
CMS100-2	+	+	+	+	237.64
CMS100-3	+	+	+	+	+
CMS100-4	+	+	+	+	+
CMS100-5	+	+	+	+	1938.22
CMS125-1	+	+	+	+	598.22
CMS125-2	+	+	+	+	+
CMS125-3	+	+	+	+	+
CMS125-4	+	+	+	+	+
CMS125-5	+	+	+	+	+
CMS150-1	+	+	+	+	+
CMS150-2	+	+	+	+	+
CMS150-3	+	+	+	+	+
CMS150-4	+	+	+	+	+
CMS150-5	+	+	+	+	+

B.3 DINS Experimental Results

We present the details of experimental results related to DINS in the following tables.

Table B.13: Percentage of gap of the solutions obtained by different solvers in one CPU-hour starting from the presumably poor solutions on 64 benchmark instances. Bold face identifies the best solver for the corresponding instance.

problem	Percentage of Gap			
	Cplex-D	LB	RINS	DINS
Small spread instances				
a1c1s1	2.347	0.250	**0.000**	0.079
a2c1s1	2.978	1.889	**0.000**	0.024
b1c1s1	5.977	1.786	**0.933**	4.444
b2c1s1	4.240	2.701	**0.559**	1.010
biella1	**0.309**	0.806	0.426	0.739
danoint	**0.000**	**0.000**	**0.000**	**0.000**
mkc	0.180	0.049	0.043	**0.021**
net12	**0.000**	**0.000**	**0.000**	**0.000**
nsrand-ipx	0.625	0.625	0.313	**0.000**
rail507	**0.000**	**0.000**	**0.000**	**0.000**
rail2586c	2.518	2.204	1.994	**1.574**
rail4284c	1.774	1.867	**1.027**	**1.027**
rail4872c	1.742	1.290	1.097	**1.032**
seymour	0.473	0.473	**0.000**	0.236
sp97ar	0.428	0.513	0.335	**0.000**
sp97ic	0.793	0.642	0.551	**0.000**
sp98ar	0.184	**0.106**	0.177	0.228

Continued on next page

Table B.13 – Continued from previous page				
problem	Percentage of Gap			
	Cplex-D	LB	RINS	DINS
sp98ic	0.270	0.146	0.204	**0.072**
tr12-30	**0.000**	0.024	**0.000**	**0.000**
arki001	0.003	0.003	0.004	**0.002**
roll3000	0.543	0.303	**0.070**	**0.070**
umts	0.013	0.049	0.022	**0.002**
berlin-5-8-0	**0.000**	**0.000**	**0.000**	**0.000**
bg512142	7.257	5.192	0.161	**0.000**
blp-ic97	0.779	0.653	0.358	**0.000**
blp-ic98	0.961	1.056	0.746	**0.515**
blp-ar98	0.655	0.060	0.461	**0.000**
cms750-4	2.372	**0.791**	1.186	**0.791**
dc11	2.018	8.166	6.994	**1.572**
railway-8-1-0	0.250	**0.000**	0.250	0.250
usabbrv-8-25-70	3.306	2.479	**0.000**	1.653
aflow40b	0.257	1.455	**0.000**	**0.000**
dano3mip	2.602	3.595	4.724	**2.230**
fast0507	**0.000**	0.575	0.575	**0.000**
harp2	0.001	0.001	0.023	**0.000**
t1717	7.948	**1.939**	5.979	7.948
noswot	**0.000**	**0.000**	**0.000**	**0.000**
timtab1	7.469	7.779	**0.000**	**0.000**
ljb2	**0.256**	3.329	1.576	3.329
rococoB10-011000	0.802	2.848	**0.437**	**0.437**
rococoB11-010000	5.039	5.839	**1.768**	2.196
rococoB12-111111	5.204	4.489	3.738	**2.541**
rococoC10-001000	0.044	0.113	0.044	**0.000**
rococoC11-011100	6.018	9.991	9.244	**5.879**
rococoC12-111100	5.188	5.188	**1.298**	4.016
Medium spread instances				
Continued on next page				

Table B.13 – Continued from previous page

problem	Percentage of Gap			
	Cplex-D	LB	RINS	DINS
glass4	13.014	7.534	**2.740**	4.794
swath	18.067	5.679	8.089	**4.622**
dg012142	17.457	25.984	4.963	**3.943**
liu	**2.475**	10.066	3.465	5.281
timtab2	16.373	18.484	3.188	**0.912**
ljb7	7.424	21.834	**4.367**	8.908
ljb9	50.717	70.866	55.074	**50.690**
ljb10	**0.807**	13.929	13.693	8.578
rococoB10-011001	7.660	5.309	**5.220**	10.082
rococoB11-110001	9.994	19.558	**4.267**	6.894
rococoC10-100001	16.041	**7.387**	13.316	10.070
rococoC11-010100	27.431	13.615	10.546	**9.029**
rococoC12-100000	12.928	10.090	5.623	**2.799**
Large spread instances				
markshare1	500.000	**400.00**	**400.00**	500.00
markshare2	1300.000	**1100.000**	2000.000	1800.000
dc1c	695.213	2.353	**0.296**	0.773
trento1	**0.000**	193.118	1.912	0.402
ds	11.226	945.745	11.226	**6.119**
ljb12	**39.273**	323.183	49.599	64.987

Table B.14: Experimental results of Cplex-D, LB, RINS, and DINS on Cornuéjols-Dawande optimality-hard instances. CD-Optxxx-y: name of the instance, where xxx and y represent n and the number of the instance respectively. Initial solution: the first solution obtained by Cplex-D, which is used as the starting solution. Entries indicate objective value of found solutions. Bold face identifies the solver which obtains the new best solution.

Instance	Objective value				
	Initial solution	Cplex-D	LB	RINS	DINS
CD-Opt40-1	587	1	1	3	3
CD-Opt40-2	512	1	1	2	2
CD-Opt40-3	642	1	2	1	2
CD-Opt40-4	982	1	1	1	**0**
CD-Opt40-5	555	2	1	1	2
CD-Opt40-6	848	1	**0**	1	**0**
CD-Opt40-7	780	1	1	1	1
CD-Opt40-8	528	2	2	2	1
CD-Opt40-9	507	1	1	1	2
CD-Opt40-10	653	1	1	**0**	3
CD-Opt50-1	926	5	3	4	4
CD-Opt50-2	815	6	5	5	6
CD-Opt50-3	667	4	3	3	5
CD-Opt50-4	790	5	5	4	5
CD-Opt50-5	946	3	6	5	5
CD-Opt50-6	1155	5	4	3	5
CD-Opt50-7	842	4	5	3	4
CD-Opt50-8	916	5	4	4	5
CD-Opt50-9	692	4	5	5	5
CD-Opt50-10	906	4	4	3	2

Continued on next page

Table B.14 – Continued from previous page

Instance	Objective value				
	Initial solution	Cplex-D	LB	RINS	DINS
CD-Opt60-1	1267	9	8	10	8
CD-Opt60-2	1301	8	8	7	11
CD-Opt60-3	952	5	9	9	5
CD-Opt60-4	822	9	9	9	9
CD-Opt60-5	1528	7	8	9	7
CD-Opt60-6	1348	10	8	8	9
CD-Opt60-7	1080	7	7	10	10
CD-Opt60-8	1289	8	7	6	11
CD-Opt60-9	722	10	8	7	10
CD-Opt60-10	1012	9	6	9	9
CD-Opt70-1	1727	13	11	16	15
CD-Opt70-2	1929	16	14	16	17
CD-Opt70-3	1713	11	12	14	14
CD-Opt70-4	1883	15	11	15	10
CD-Opt70-5	1576	14	12	17	16
CD-Opt70-6	1902	13	14	16	17
CD-Opt70-7	1734	13	13	11	18
CD-Opt70-8	1522	12	10	13	18
CD-Opt70-9	1580	16	11	14	15
CD-Opt70-10	1442	10	13	15	15
CD-Opt80-1	2081	19	20	23	23
CD-Opt80-2	2496	23	27	14	22
CD-Opt80-3	1602	21	18	21	23
CD-Opt80-4	2230	19	24	21	24
CD-Opt80-5	1904	19	19	20	23
CD-Opt80-6	1686	13	23	21	24

Continued on next page

Table B.14 – Continued from previous page

Instance	Objective value				
	Initial solution	Cplex-D	LB	RINS	DINS
CD-Opt80-7	2196	19	18	21	21
CD-Opt80-8	1492	20	18	24	24
CD-Opt80-9	1798	20	15	23	23
CD-Opt80-10	2143	22	17	25	26
CD-Opt90-1	2467	27	33	30	30
CD-Opt90-2	3287	34	31	30	33
CD-Opt90-3	2183	32	29	34	30
CD-Opt90-4	2177	24	25	26	33
CD-Opt90-5	2494	33	25	28	26
CD-Opt90-6	2531	17	29	33	35
CD-Opt90-7	3335	35	34	29	30
CD-Opt90-8	1796	20	27	28	17
CD-Opt90-9	2547	37	30	34	29
CD-Opt90-10	1919	26	23	28	23
CD-Opt100-1	3210	39	55	31	35
CD-Opt100-2	3605	40	44	38	35
CD-Opt100-3	3461	40	48	46	42
CD-Opt100-4	3522	33	48	34	40
CD-Opt100-5	2575	48	43	40	42
CD-Opt100-6	4122	46	47	43	40
CD-Opt100-7	3085	41	41	41	42
CD-Opt100-8	3173	45	43	46	42
CD-Opt100-9	3473	47	46	40	27
CD-Opt100-10	3157	50	42	40	43

Table B.15: Experimental results of Cplex-D, LB, RINS, and DINS on constrained market-sharing instances with $k = 2.0$. CMSxxx-y: name of the instance, where xxx and y represent n and the number of the instance respectively. Initial solution: the first solution obtained by PGC_1, which is used as the starting solution. Entries indicate objective value of found solutions. Bold face identifies the solver which obtains the new best solution.

Instance	Objective value				
	Initial solution	Cplex-D	LB	RINS	DINS
CMS50-1	1381	1381	1381	830	**768**
CMS50-2	1890	1890	1890	**803**	**803**
CMS50-3	1798	1798	1798	918	**870**
CMS50-4	1996	1996	1996	**856**	904
CMS50-5	1638	1638	1638	**661**	820
CMS50-6	1681	1681	1111	923	**874**
CMS50-7	1960	1960	1960	898	**841**
CMS50-8	2127	2127	2127	880	**838**
CMS50-9	1481	1481	1285	1005	**873**
CMS50-10	1421	1421	1421	**701**	729
CMS100-1	4747	4747	4747	4747	4747
CMS100-2	5667	5667	5667	5667	**4075**
CMS100-3	4303	4303	4303	4303	4303
CMS100-4	4846	4846	4846	4846	4846
CMS100-5	5047	5047	5047	5047	5047
CMS100-6	5483	5483	5483	5483	5483
CMS100-7	4814	4814	4814	4814	4814
CMS100-8	5462	5462	5462	5462	**3381**
CMS100-9	4625	4625	4625	4625	4625
CMS100-10	5374	5374	5374	5374	5374

Continued on next page

Table B.15 – Continued from previous page					
Name	Initial solution	Cplex-D	LB	RINS	DINS
CMS150-1	10611	10611	10611	10611	10611
CMS150-2	9618	9618	9618	9618	9618
CMS150-3	9416	9416	9416	9416	9416
CMS150-4	8186	8186	8186	8186	8186
CMS150-5	9652	9652	9652	9652	9652
CMS150-6	8613	8613	8613	8613	8613
CMS150-7	9188	9188	9188	9188	9188
CMS150-8	8995	8995	8995	8995	8995
CMS150-9	9174	9174	9174	9174	9174
CMS150-10	9290	9290	9290	9290	9290

Table B.16: Experimental results of Cplex-D, LB, RINS, and DINS on constrained market-sharing instances with $k = 1.5$. CMSxxx-y: name of the instance, where xxx and y represent n and the number of the instance respectively. Initial solution: the first solution obtained by PGC_1, which is used as the starting solution. Entries indicate objective value of found solutions. Bold face identifies the solver which obtains the new best solution.

Instance	Objective value				
	Initial solution	Cplex-D	LB	RINS	DINS
CMS50-1	2079	2079	2079	1845	**1804**
CMS50-2	2530	2530	2530	2057	**1716**
CMS50-3	2296	2296	2296	2197	**1413**
CMS50-4	2424	2424	2424	2424	**1715**
CMS50-5	1880	1880	1880	1880	**1769**
Continued on next page					

Table B.16 – Continued from previous page

Name	Initial solution	Cplex-D	LB	RINS	DINS
CMS50-6	2374	2374	2374	**1848**	1877
CMS50-7	2507	2507	2507	1917	**1533**
CMS50-8	2499	2499	2499	1727	**1710**
CMS50-9	2778	2778	2778	**1562**	1991
CMS50-10	2170	2170	2170	1731	**1649**
CMS75-1	4886	4886	4886	4886	**3999**
CMS75-2	4233	4233	4233	4233	**2993**
CMS75-3	4095	4095	4095	4095	**3531**
CMS75-4	4313	4313	4313	4313	**3263**
CMS75-5	5096	5096	5096	5096	**3564**
CMS75-6	4489	4489	4489	4489	**3701**
CMS75-7	3855	3855	3855	3855	**3247**
CMS75-8	4342	4342	4342	4342	**3336**
CMS75-9	4556	4556	4556	4556	**3963**
CMS75-10	4284	4284	4284	4284	4284
CMS100-1	6742	6742	6742	6742	6742
CMS100-2	7311	7311	7311	7311	7311
CMS100-3	6571	6571	6571	6571	6571
CMS100-4	6030	6030	6030	6030	6030
CMS100-5	6465	6465	6465	6465	**5334**
CMS100-6	6073	6073	6073	6073	6073
CMS100-7	7921	7921	7921	7921	7921
CMS100-8	6433	6433	6433	6433	6433
CMS100-9	7368	7368	7368	7368	7368
CMS100-10	6023	6023	6023	6023	6023

Table B.17: Experimental results of Cplex-D, LB, RINS, and DINS on constrained market-sharing instances with $k = 1.3$. Entries indicate objective value of found solutions. Initial solution: the first solution obtained by PGC_1, which is used as the starting solution. CMSxxx-y: name of the instance, where xxx and y represent n and the number of the instance respectively. Bold face identifies the solver which obtains the new best solution.

Instance	Objective value				
	Initial solution	Cplex-D	LB	RINS	DINS
CMS50-1	3197	3197	3197	3197	**2226**
CMS50-2	2283	2283	2283	2283	2283
CMS50-3	2924	2924	2924	2924	**2216**
CMS50-4	2896	2896	2788	2896	**2149**
CMS50-5	2495	2495	2495	2495	2495
CMS50-6	3138	3138	**2253**	2261	**2253**
CMS50-7	2785	2785	2785	2785	**2704**
CMS50-8	2439	2439	2439	2439	**2151**
CMS50-9	2590	2590	2590	2590	**2496**
CMS50-10	2427	2427	2169	2169	**2074**
CMS75-1	4639	4639	4639	4639	4639
CMS75-2	5900	5900	5900	5900	**5116**
CMS75-3	4850	4850	4850	4850	**4546**
CMS75-4	4684	4684	4684	4684	**4669**
CMS75-5	5267	5267	5267	5267	5267
CMS75-6	4858	4858	4858	4858	4858
CMS75-7	4668	4668	4668	4668	4668
CMS75-8	4792	4792	4792	4792	**4661**
CMS75-9	5263	5263	5263	5263	5263
CMS75-10	4426	4426	4426	4426	4426

Continued on next page

Table B.17 – Continued from previous page					
Name	Initial solution	Cplex-D	LB	RINS	DINS
CMS100-1	8687	8687	8687	8687	8687
CMS100-2	7218	7218	7218	7218	7218
CMS100-3	7463	7463	7463	7463	7463
CMS100-4	8344	8344	8344	8344	8344
CMS100-5	7884	7884	7884	7884	7884
CMS100-6	7861	7861	7861	7861	7861
CMS100-7	8448	8448	8448	8448	8448
CMS100-8	6969	6969	6969	6969	6969
CMS100-9	8029	8029	8029	8029	8029
CMS100-10	7585	7585	7585	7585	7585

Table B.18: RINS neighbourhoods versus DINS neighbourhoods. NN: the number of explored neighbourhoods, ER: the average enumeration ratio.

problem	RINS		DINS		problem	RINS		DINS	
	NN	ER	NN	ER		NN	ER	NN	ER
a1c1s1	219	0.571	252	0.276	bg512142	266	0.633	473	0.390
a2c1s1	204	0.593	216	0.294	blp-ic97	1038	0.010	1432	0.005
b1c1s1	86	0.598	101	0.243	blp-ic98	911	0.006	1477	0.003
b2c1s1	63	0.636	84	0.269	blp-ar98	756	0.011	1163	0.006
biella1	134	0.084	240	0.030	cms750-4	627	0.156	476	0.110
danoint	321	0.652	343	0.471	dc1c	56	0.063	89	0.028
glass4	27595	0.223	13237	0.206	dc1l	29	0.015	56	0.005
markshare1	50956	0.490	47290	0.642	dg012142	137	0.449	222	0.187
markshare2	45069	0.452	43875	0.535	railway-8-1-0	5029	0.248	4071	0.173
mkc	429	0.048	819	0.027	trento1	77	0.074	225	0.022

Continued on next page

Table B.18 – Continued from previous page

problem	RINS		DINS		problem	RINS		DINS	
	NN	ER	NN	ER		NN	ER	NN	ER
net12	14	0.449	28	0.195	usabbrv-8-25-70	3792	0.315	4216	0.241
nsrand-ipx	1293	0.017	934	0.012	aflow40b	3400	0.042	2561	0.041
rail507	77	0.004	78	0.002	dano3mip	3	0.243	7	0.064
rail2586c	22	0.099	47	0.025	ds	37	0.014	55	0.003
rail4284c	17	0.072	52	0.021	fast0507	24	0.005	76	0.002
rail4872c	11	0.083	57	0.022	harp2	17010	0.029	21697	0.026
seymour	82	0.328	134	0.109	liu	1503	0.330	431	0.273
sp97ar	436	0.013	667	0.007	t1717	168	0.008	198	0.002
sp97ic	1521	0.006	2131	0.004	ljb2	2239	0.091	1832	0.069
sp98ar	766	0.010	1087	0.006	ljb7	297	0.048	160	0.035
sp98ic	1029	0.007	1817	0.004	ljb9	250	0.061	73	0.044
swath	5864	0.010	5510	0.008	ljb10	174	0.080	126	0.045
tr12-30	4035	0.217	1618	0.168	ljb12	195	0.086	101	0.047
berlin-5-8-0	1605	0.267	3605	0.233					

B.4 NPGC Experimental Results

We present the details of experimental results related to NPGC in the following table.

Table B.19: Percentage of gap of the solutions obtained by different solvers in one CPU-hour on 53 0-1 mixed integer program benchmark instances. Bold face identifies the best method for the corresponding instance. + : a time limit of 1 CPU-hour exceeded.

problem	Cplex-D	LB	RINS	NPGC
Small spread instances				
a1c1s1	2.057	0.077	**0.017**	0.768
a2c1s1	2.978	1.889	**0.000**	0.746
b1c1s1	5.977	7.305	**0.933**	3.242
b2c1s1	4.240	2.701	**0.559**	6.017
biella1	0.368	0.272	**0.005**	0.232
danoint	**0.000**	**0.000**	**0.000**	**0.000**
mkc	0.344	0.148	**0.000**	2.518
nsrand-ipx	**0.313**	0.625	**0.313**	**0.313**
rail507	**0.000**	0.575	**0.000**	**0.000**
rail2586c	2.518	**1.154**	1.994	2.099
rail4284c	1.774	2.334	**1.027**	1.587
rail4872c	1.613	**0.581**	1.355	1.161
seymour	0.473	0.236	**0.000**	0.236
sp97ar	0.544	**0.109**	0.260	0.580
sp97ic	0.793	0.642	**0.551**	1.120
sp98ar	0.218	**0.023**	0.177	0.091
sp98ic	0.222	0.172	**0.090**	0.366
tr12-30	0.009	0.047	**0.000**	0.080
berlin-5-8-0	**0.000**	**0.000**	**0.000**	**0.000**
bg512142	7.939	5.192	**0.502**	0.580
blp-ic97	1.203	**0.152**	0.640	1.879

Continued on next page

Table B.19 – Continued from previous page

problem	Cplex-D	LB	RINS	NPGC
blp-ic98	**0.823**	1.259	1.492	4.901
blp-ar98	0.407	0.343	0.593	**0.197**
cms750-4	1.581	1.186	**0.395**	0.791
railway-8-1-0	**0.000**	**0.000**	**0.000**	**0.000**
usabbrv-8-25-70	0.826	4.132	0.826	**0.000**
aflow40b	0.257	**0.000**	1.455	**0.000**
dano3mip	**1.220**	3.595	4.724	3.214
fast0507	**0.000**	0.575	0.575	0.575
harp2	0.002	0.001	0.010	**0.000**
t1717	5.979	6.449	5.979	**0.427**
ljb2	3.329	3.329	**0.039**	7.564
Medium spread instances				
glass4	13.014	8.733	**2.740**	7.306
swath	18.067	**5.679**	8.089	17.305
dg012142	16.611	10.904	1.596	**6.449**
liu	3.300	10.066	1.980	**0.000**
ljb7	23.581	21.834	**4.367**	44.192
ljb9	61.529	70.866	**55.074**	58.701
ljb10	**4.013**	18.690	13.693	36.238
Large spread instances				
markshare1	500.000	**400.000**	**400.000**	2400.000
markshare2	1300.000	**1100.000**	2000.000	4200.000
dc1c	821.785	2.353	656.689	**1.467**
trento1	3.800	193.094	**0.008**	3.792
ds	**11.226**	1016.086	**11.226**	29.168
ljb12	**30.426**	398.596	42.506	72.531
net12	**0.000**	**0.000**	**0.000**	+
dc1l	**2.018**	8.776	969.933	+
momentum1	+	+	+	+
protfold	+	+	+	**70.000**

Continued on next page

Table B.19 – Continued from previous page

problem	Cplex-D	LB	RINS	NPGC
rd-rplusc-21	+	+	+	**0.000**
nsr8k	+	+	+	+
dolom1	+	+	+	+
siena1	+	+	+	+